

A Software Framework for Data Based Analysis

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum politicarum
(Doktor der Wirtschaftswissenschaft)
im Fach Volkswirtschaft

eingereicht an der
Wirtschaftswissenschaftlichen Fakultät
Humboldt-Universität zu Berlin

von

Dipl.-Vw. Markus Krätzig
geboren am 16.11.1974 in Berlin

Präsident der Humboldt-Universität zu Berlin:

Prof. Dr. Jürgen Mlynek

Dekan der Wirtschaftswissenschaftlichen Fakultät:

Prof. Dr. Joachim Schwalbach

Gutachter:

1. Prof. Dr. Helmut Lütkepohl

2. Prof. Dr. Bettina Berendt

eingereicht am: 21.12.2004

Tag der mündlichen Prüfung: 04.02.2005

Zusammenfassung

Es wird das Software Framework *JStatCom* vorgestellt, welches die Entwicklung von leistungsfähigen grafischen Benutzerschnittstellen für Daten-basierte Analysemethoden wesentlich vereinfacht, wobei der Schwerpunkt auf Methoden der Ökonometrie, insbesondere der Zeitreihenanalyse liegt. Das Konzept besteht darin, sämtliche wiederkehrenden Aufgaben mit Hilfe von Java-Klassen zu lösen, sowie die Ausführung von speziellen Algorithmen an externe Programme, wie z.B. Gauss oder Matlab, zu delegieren. Auf diese Weise können schon existierende Prozeduren aus verschiedenen Programmiersprachen wiederverwendet werden. Weiterhin wird die ökonomische Anwendungssoftware *JMulti* beschrieben, die auf Basis dieses Frameworks erstellt wurde.

Schlagwörter:

Zeitreihenanalyse, Wissenschaftliches Rechnen, Software Entwicklung, JMulti

Abstract

This work presents the software framework *JStatCom* which is geared towards the development of powerful graphical user interfaces for data based analysis methods, especially for econometrics and time series analysis. The concept is to solve all recurring tasks with the help of Java classes and to delegate the execution of special algorithms to external programs, for example Gauss or Matlab. This way it is possible to reuse already existing procedures written in different programming languages. Furthermore, the econometric software *JMulti* will be presented which has been developed with the help of this framework.

Keywords:

time series analysis, scientific computing, software engineering, JMulti

Table of Contents

List of Figures	xvi
List of Tables	xviii
Abbreviations	xix
Acknowledgements	xxi
1 From Algorithms towards an integrated Framework	1
1.1 Problem Description	1
1.2 JStatCom and JMulTi	3
1.3 Access to Algorithms	7
1.3.1 Using an Execution Engine	7
1.3.2 Using Libraries and Toolkits	9
1.3.3 Including Graphics	11
1.3.4 Integrating available Tools	13
1.4 JStatCom in comparison to other Approaches	14
1.4.1 MulTi	14
1.4.2 Observations on Entropy and Software Reuse	14
1.4.3 The MMM Project	16
1.4.4 The OmegaHat Project	18
1.4.5 Creating GUI's with Matlab	19
1.4.6 Creating interactive Programs with Ox	20
1.5 Concluding Remarks	23
1.6 How to read this Thesis	24

2	A Motivating Example	26
2.1	Introduction	26
2.2	A Step-by-Step Example	27
2.2.1	System Requirements for this Example	27
2.2.2	Step 1: Download/Install the Java Software Development Kit (J2SE SDK)	28
2.2.3	Step 2: Download/Install the Eclipse IDE and some Plug-ins	28
2.2.4	Step 3: Download JStatCom and unpack it	29
2.2.5	Step 4: Create an Eclipse Project	30
2.2.6	Step 5: Create a new Package and a new Class	31
2.2.7	Step 6: Layout the GUI with the Visual Editor	36
2.2.8	Step 7: Add an Action to the Execute Button	45
2.2.9	Step 8: Add the Module to the Main Application Frame .	45
2.2.10	Step 9: Integrate Gauss Algorithm	47
2.2.11	Step 10: Implement the Execute Routine	50
2.2.12	Step 11: Check running Module	55
2.3	Final Remarks	56
3	Design and Implementation	58
3.1	Documenting a Software Architecture	58
3.2	JStatCom System Overview	59
3.3	How Stakeholders Can Use the Documentation	60
3.3.1	Typical Development Steps	61
3.4	Background, Rationale, and Design Constraints	63
3.4.1	Operational Context	63
3.4.2	Key Data Management Features	64
3.4.3	Key User Interface Features	64
3.4.4	Key Interoperability Features	65
3.4.5	Key Design Features	66
3.5	JStatCom Architecture View Template	69
3.6	View Packet 1: JStatCom	70
3.6.1	Primary Presentation	70
3.6.2	Element Catalog	72
3.6.3	Context	74

3.6.4	Architecture Background	76
3.6.5	Related View Packets	76
3.7	View Packet 2: Data Model	76
3.7.1	Primary Presentation	76
3.7.2	Element Catalog	77
3.7.3	Context	78
3.7.4	Architecture Background	79
3.7.5	Related View Packets	79
3.8	View Packet 3: Type System	79
3.8.1	Primary Presentation	79
3.8.2	Element Catalog	82
3.8.3	Architecture Background	83
3.8.4	Usage Example	85
3.8.5	Related View Packets	86
3.9	View Packet 4: Data Event System	86
3.9.1	Primary Presentation	86
3.9.2	Element Catalog	87
3.9.3	Architecture Background	88
3.9.4	Usage Example	89
3.9.5	Related View Packets	91
3.10	View Packet 5: Symbol Management	91
3.10.1	Primary Presentation	91
3.10.2	Element Catalog	94
3.10.3	Architecture Background	95
3.10.4	Usage Example	96
3.10.5	Related View Packets	100
3.11	View Packet 6: Symbol Event System	101
3.11.1	Primary Presentation	101
3.11.2	Element Catalog	102
3.11.3	Architecture Background	103
3.11.4	Usage Example	104
3.11.5	Related View Packets	106
3.12	View Packet 7: Symbol Control	106
3.12.1	Primary Presentation	106

3.12.2	Element Catalog	110
3.12.3	Architecture Background	111
3.12.4	Usage Example	111
3.12.5	Related View Packets	112
3.13	View Packet 8: Engine	113
3.13.1	Primary Presentation	113
3.13.2	Element Catalog	119
3.13.3	Context	121
3.13.4	Architecture Background	122
3.13.5	Usage Example	125
3.13.6	Related View Packets	125
3.14	View Packet 9: Gauss	126
3.14.1	Primary Presentation	126
3.14.2	Element Catalog	131
3.14.3	Architecture Background	132
3.14.4	Usage Example	133
3.14.5	Related View Packets	135
3.15	View Packet 10: GRTE	135
3.15.1	Primary Presentation	135
3.15.2	Element Catalog	140
3.15.3	Architecture Background	140
3.15.4	Usage Example	143
3.15.5	Related View Packets	143
3.16	View Packet 11: Ox	143
3.16.1	Primary Presentation	143
3.16.2	Element Catalog	145
3.16.3	Architecture Background	146
3.16.4	Usage Example	149
3.16.5	Related View Packets	150
3.17	View Packet 12: Stub	151
3.17.1	Primary Presentation	151
3.17.2	Element Catalog	154
3.17.3	Architecture Background	155
3.17.4	Usage Example	156

3.17.5	Related View Packets	157
3.18	View Packet 13: MatLab	158
3.18.1	Primary Presentation	158
3.18.2	Element Catalog	160
3.18.3	Architecture Background	160
3.18.4	Usage Example	161
3.18.5	Related View Packets	162
3.19	View Packet 14: PCall	162
3.19.1	Primary Presentation	162
3.19.2	Element Catalog	164
3.19.3	Architecture Background	165
3.19.4	Usage Example	167
3.19.5	Related View Packets	169
3.20	View Packet 15: Time Series	170
3.20.1	Primary Presentation	170
3.20.2	Element Catalog	171
3.20.3	Context	173
3.20.4	Architecture Background	173
3.20.5	Usage Example	176
3.20.6	Related View Packets	177
3.21	View Packet 16: List	177
3.21.1	Primary Presentation	177
3.21.2	Element Catalog	180
3.21.3	Architecture Background	182
3.21.4	Usage Example	183
3.21.5	Related View Packets	184
3.22	View Packet 17: Table	184
3.22.1	Primary Presentation	184
3.22.2	Element Catalog	186
3.22.3	Architecture Background	186
3.22.4	Related View Packets	187
3.23	View Packet 18: Selection	187
3.23.1	Primary Presentation	187
3.23.2	Element Catalog	190

3.23.3	Architecture Background	190
3.23.4	Usage Example	191
3.23.5	Related View Packets	192
3.24	View Packet 19: Calculator	192
3.24.1	Primary Presentation	192
3.24.2	Element Catalog	194
3.24.3	Architecture Background	195
3.24.4	Usage Example	195
3.24.5	Related View Packets	196
3.25	View Packet 20: Components	196
3.25.1	Primary Presentation	196
3.25.2	Element Catalog	197
3.25.3	Context	200
3.25.4	Architecture Background	200
3.25.5	Usage Example	201
3.25.6	Related View Packets	204
3.26	View Packet 21: Application	204
3.26.1	Primary Presentation	204
3.26.2	Element Catalog	208
3.26.3	Architecture Background	209
3.26.4	Usage Example	213
3.26.5	Related View Packets	215
3.27	View Packet 22: Data Table	215
3.27.1	Primary Presentation	215
3.27.2	Element Catalog	219
3.27.3	Architecture Background	220
3.27.4	Usage Example	221
3.27.5	Related View Packets	223
3.28	View Packet 23: Equation	223
3.28.1	Primary Presentation	223
3.28.2	Element Catalog	225
3.28.3	Architecture Background	226
3.28.4	Usage Example	227
3.28.5	Related View Packets	229

3.29	View Packet 24: Input/Output	230
3.29.1	Primary Presentation	230
3.29.2	Element Catalog	232
3.29.3	Architecture Background	233
3.29.4	Related View Packets	234
3.30	View Packet 25: Data Import System	235
3.30.1	Primary Presentation	235
3.30.2	Element Catalog	238
3.30.3	Architecture Background	239
3.30.4	Usage Example	240
3.30.5	Related View Packets	241
3.31	Concluding Remarks about JStatCom	241
4	JMulTi - A Reference Application of the Framework	245
4.1	Introduction	245
4.2	General Setup	246
4.3	Modules of JMulTi	248
4.4	How to read this Chapter	250
4.5	Initial Analysis	250
4.5.1	Overview	250
4.5.2	Implemented Features	252
4.5.3	Implementation Details	256
4.6	VAR Analysis	261
4.6.1	Overview	261
4.6.2	Implemented Features	265
4.6.3	Implementation Details	272
4.7	VEC Analysis	286
4.7.1	Overview	286
4.7.2	Implemented Features	290
4.7.3	Implementation Details	295
4.8	ARCH Analysis	299
4.8.1	Overview	299
4.8.2	Implemented Features	301
4.8.3	Implementation Details	304

4.9	STR Analysis	305
4.9.1	Overview	305
4.9.2	The Modelling Cycle	306
4.9.3	Implemented Features	307
4.9.4	Implementation Details	315
4.10	Nonparametric Analysis	316
4.10.1	Overview	316
4.10.2	Implemented Features	319
4.10.3	Implementation Details	324
4.11	Outlook	328
4.12	Conclusion	329
A	Guide to Notation	331
A.1	Class Diagrams	332
A.1.1	Elements and Inheritance	332
A.1.2	Components	334
A.1.3	Relations between Elements of Class Diagrams	335
A.2	Object Diagrams	337
A.3	Use Case Diagrams	337
B	Documenting Modules with JavaHelp and JHelpDev	339
B.1	LaTeX and latex2html	340
B.2	JHelpDev	341
B.3	Integrating Helpsets with an Application	343
	Bibliography	353
	Selbständigkeitserklärung	354

List of Figures

1.1	A screenshot of a simple example module for Markov-Switching models	4
1.2	Class diagram for an interactive Ox program	21
2.1	Selecting a new Java project in the Eclipse IDE	30
2.2	Setting project name and directory	31
2.3	Display of the new project <code>testproject</code> in the package explorer	32
2.4	Creating a new class	33
2.5	Creating a new package	34
2.6	Specifying the new class	34
2.7	Generated Java code for a new class	35
2.8	Open class in visual builder tool	36
2.9	Initial display, frame needs to be resized	37
2.10	<code>TestFrame</code> after resize with title, properties at the bottom	38
2.11	<code>TestFrame</code> after a <code>JPanel</code> has been added	39
2.12	Setting layout manager to <code>null</code> , easier to handle for initial design	40
2.13	Selecting the <code>TSSel</code> component	41
2.14	<code>TSSel</code> component has been placed on the panel	42
2.15	Editing properties of <code>TSSel</code> component	43
2.16	Selecting the <code>NumSelector</code> component for number input	44
2.17	Setting a validating range <code>[1, 20]</code> to the <code>NumSelector</code> component	44
2.18	Placing a <code>JButton</code> and creating an action	45
2.19	Default action handler for execute button	46
2.20	Editing <code>modules.xml</code> to insert <code>TestFrame</code> to list of modules	47
2.21	Adjusting the classpath in the <code>app.bat</code> script	48

2.22	Running application with new <code>TestFrame</code> module, execute method still needs to be coded	49
2.23	Open <code>TestFrame</code> in Java editor, more convenient for manual coding than visual editor	51
2.24	Hidden method body that should be expanded by clicking on arrow to the left	52
2.25	Red underline and symbol to the left indicating some compilation problem	52
2.26	A left mouse click on the error symbol gives a menu with possible error fixes, the first option should be chosen here	53
2.27	Implementation of the execute call	54
2.28	Running module with output from computation	55
2.29	Symbol Control after the computation has finished	56
3.1	Use cases for <code>JStatCom</code>	75
3.2	Context of the Data Model	78
3.3	Type System	80
3.4	Classes in Data Event System	86
3.5	Symbol Management	92
3.6	<code>SymbolScope</code> inheritance	93
3.7	Class structure of a hypothetical VAR frame	97
3.8	Snapshot of model objects and shared data with different scopes	98
3.9	Classes in Symbol Event System	101
3.10	Tree related classes in Symbol Control	107
3.11	Snapshot of objects in symbol tree	108
3.12	GUI related classes in Symbol Control	109
3.13	Screenshot of symbol frame with selected <code>NARRAY</code>	112
3.14	Screenshot of symbol frame with selected <code>DRANGE</code>	113
3.15	Engine classes	114
3.16	Engine inheritance	115
3.17	<code>ConfigDialog</code> with a <code>DefaultConfigPanel</code>	116
3.18	Engine client using abstract <code>Engine</code> and <code>EngineTypes</code> , but specific <code>LoadTypes</code>	118
3.19	Context of the Engine system	122

3.20	Classes of the Gauss subsystem	126
3.21	Gauss communication libraries	127
3.22	Classes of the GRTE subsystem	135
3.23	Classes of the Ox subsystem	143
3.24	Classes of the Stub subsystem	151
3.25	Classes of the MatLab subsystem	158
3.26	Classes of the PCall subsystem	163
3.27	Classes of the Time Series subsystem	171
3.28	Context of the Time Series subsystem	174
3.29	Classes of the List subsystem	178
3.30	Screenshot of a TSList component with TSListPopup showing .	179
3.31	Classes of the Table subsystem	184
3.32	Screenshot of a TSTable component	185
3.33	Screenshot of a TSSel component	187
3.34	Screenshot of Time Series Calculator	192
3.35	Classes for input validating text fields	197
3.36	Screenshot of NumSelector with an error message	201
3.37	Use cases for the Components system	202
3.38	Specification of a NumSelector in a visual application builder . .	203
3.39	Classes of Application system	205
3.40	Screenshot of TopFrame customized for the JMulti application .	206
3.41	Classes of Data Table system	215
3.42	Screenshot of a NArrayTable	216
3.43	Screenshot of NArrayTable with a special table renderer and mouse click listener	216
3.44	Screenshot of a SArrayTable	217
3.45	Classes of Equation system	223
3.46	Screenshot of VEC model coefficients estimates	224
3.47	Screenshot of VEC model exclusion restrictions on shortrun dy- namics	224
3.48	Classes of Input/Output system	230
3.49	Error message presented to the user	230
3.50	Screenshot of LogFrame with a detailed error message	231
3.51	Classes of Data Import system	235

3.52	Screenshot of <code>ImportDataFrame</code> with a <code>TSImportPanel</code>	236
3.53	Screenshot of <code>TSASCIIDialog</code>	237
4.1	Screenshot of workbench with autocorrelation panel	251
4.2	Screenshot of ADF unit root test panel	252
4.3	Screenshot of Johansen cointegration test panel	253
4.4	Screenshot of Symbol Control for Initial Analysis	258
4.5	Classes for Initial Analysis	259
4.6	Screenshot of specification panel for the VAR analysis	261
4.7	Screenshot of estimation panel for the VAR analysis	262
4.8	Screenshot of manual/automatic subset specification for the VAR analysis	263
4.9	Screenshot of SVAR model estimation	263
4.10	Screenshot of Symbol Control system for VAR model	264
4.11	Screenshot of panel for plotting recursive coefficients estimates	267
4.12	Screenshot of panel for computing bootstrapped Chow tests	268
4.13	Screenshot of forecast panel for VAR analysis	269
4.14	Screenshot of bootstrap specification panel for VAR Impulse Re- sponse Analysis	270
4.15	Screenshot of VAR Impulse Response Analysis panel	270
4.16	Screenshot of VEC model selection	286
4.17	Screenshot of VEC estimation output in matrix form	287
4.18	Screenshot of specifying restrictions on short-run dynamics for a VEC model	288
4.19	Screenshot of specifying restrictions on the cointegration relation of a VEC model	289
4.20	Screenshot of the dialog for specifying the estimation of the 1st stage of a two stage VEC estimation procedure	290
4.21	Screenshot of panel for plotting recursive coefficients estimates	291
4.22	Screenshot of causality tests panel	293
4.23	Screenshot of specification panel for SVEC estimation	294
4.24	Screenshot of a user message about long-run restrictions not being taken into account with the currently selected estimation procedure	296
4.25	Screenshot of ARCH analysis module	300

4.26	Screenshot of output for univariate GARCH(1, 1) estimation . . .	301
4.27	Screenshot of residual analysis for ARCH analysis	302
4.28	Screenshot of output for multivariate GARCH(1, 1) estimation . .	303
4.29	Screenshot of model selection for STR analysis	305
4.30	Screenshot of selecting subset restrictions for AR parts of STR model	307
4.31	Screenshot of test for nonlinearity	308
4.32	Screenshot of grid search to find starting values	310
4.33	Screenshot of panel for STR estimation	311
4.34	Screenshot of dialog to set restrictions for STR estimation	312
4.35	Screenshot of graphical analysis panel	312
4.36	Screenshot of residual analysis panel	314
4.37	Screenshot of model selection for the nonparametric analysis . . .	317
4.38	Screenshot of model estimation for the nonparametric analysis . .	319
4.39	Screenshot of forecasts for the nonparametric analysis	320
4.40	Screenshot of surface plot for conditional mean	321
4.41	Screenshot of conditional mean together with Bonferroni CIs . . .	322
4.42	Screenshot of text output after lag selection finished	328
A.1	Classes and inheritance	332
A.2	Components	334
A.3	Relations among classes	335
A.4	Object diagram	337
A.5	Use case diagram	338
B.1	Screenshot of a TOC editor component	342
B.2	Screenshot of a help viewer	343

List of Tables

3.1	Typical development steps	63
3.2	Primary Presentation of JStatCom	71
3.3	Resources for MatLab engine, Windows	72
3.4	Elements of JStatCom	74
3.5	Primary Presentation of Data Model	77
3.6	Elements of the Data Model	78
3.7	Elements of the Type System	83
3.8	Elements of the Data Event System	88
3.9	Elements of the Symbol Management	95
3.10	Elements of the Symbol Event System	102
3.11	Elements of the Symbol Control	110
3.12	Primary Presentation of Engine system	113
3.13	Elements of Engine system	121
3.14	Resources for Gauss engine	130
3.15	Elements of Gauss system	131
3.16	Resources for GRTE engine, Windows	138
3.17	Resources for GRTE engine, Linux/Solaris	140
3.18	Elements of GRTE system	140
3.19	Resources for Ox engine, all operating systems	145
3.20	Elements of Ox system	145
3.21	Type conversion between JStatCom and Ox	148
3.22	Type conversion between C-types and Stub engine, all properties that are not mentioned must not be set or set to false	153
3.23	Resources for Stub engine, Windows and Linux/Solaris	154
3.24	Elements of Stub system	155

3.25	Resources for MatLab engine	159
3.26	Elements of MatLab system	160
3.27	Elements of PCall system	164
3.28	Subsystems of Time Series	170
3.29	Elements of Time Series system	173
3.30	Elements of List system	182
3.31	Elements of Table system	186
3.32	Elements of Selection system	190
3.33	Elements of Calculator system	194
3.34	Primary Presentation of Components	196
3.35	Elements of Components system	200
3.36	Resources for the Application system	207
3.37	Elements of Application system	208
3.38	Elements of Data Table system	220
3.39	Elements of Equation system	226
3.40	Elements of Input/Output system	233
3.41	Elements of Data Import system	239
4.1	Modules of JMulTi	249
4.2	Shared components	299

Abbreviations

AC	Autocorrelation
ARCH	Autoregressive Conditional Heteroskedasticity
API	Application Programming Interface
CI	Confidence Interval
DLL	Dynamic Link Library
EC	Error Correction
GLS	Generalized Least Squares
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JNI	Java Native Interface
JRE	Java Runtime Environment
LGPL	Lesser General Public License
LM	Lagrange Multiplier
ML	Maximum Likelihood
OS	Operating System
PAC	Partial Autocorrelation
PDF	Portable Document Format
STR	Smooth Transition Regression
SVAR	Structural Vector Autoregression
UML	Unified Modelling Language
UR	Unit Root
URL	Uniform Resource Locator
VAR	Vector Autoregression
VE	Visual Editor

VEC	Vector Error Correction
VB	Visual Basic
XML	Extendable Markup Language
XP	Extreme Programming

Acknowledgements

The early stages of this work were carried out in the PhD program of “Applied Microeconomics” of the Humboldt-Universität zu Berlin and completed at the the Sonderforschungsbereich 373 (SFB 373) as well as in the Chair of Econometrics. I am thankful for financial support granted by the German Research Foundation (DFG), as well as for the excellent work conditions provided by the School of Business and Economics of the Humboldt-Universität zu Berlin.

I would like to point out that this work would not have been possible without the continued support of my supervisor, Prof. Dr. Helmut Lütkepohl, who always provided valuable feedback and comments which contributed greatly to the final outcome. He was the first user of the developed software and thus assisted with its birth and its introduction to a wider audience. In addition, Prof. Bettina Berendt assisted me to improve the thesis by providing the perspectives of a computer scientist, not an econometrician. I thank her for her time and consideration.

Many econometricians have contributed their code to this system and I would like to thank, in random order, Prof. Rolf Tschernig, Dr. Ralf Brüggemann, Dr. Carsten Trenkler, Prof. Helmut Herwartz, Dmitri Boreiko, Christian Kascha, Stefan Lundbergh (Ph.D.), and Markku Lanne (Ph.D.) for their support and the fruitful collaboration. Among the participants of a particularly instructive workshop held in March 2003 in Florence were Dr. Kirstin Hubrich, Prof. Jörg Breitung, Maria Eleftheriou, Aaron Mehrotra, and Sebastian Watzka. I would also like to thank them for giving a profound feedback to the author.

This thesis is based on previous work by Dr. Alexander Benkwitz, who initiated this project and created the first prototypes. I greatly enjoyed working with him and due to the open exchange of results and knowledge I was able to continue the project after his departure from the SFB 373.

I would also like to thank Prof. Timo Teräsvirta for his comments upon the research after he kindly invited me for a stay at the Stockholm School of Economics. Last, but certainly not least, I greatly appreciated the 8-months of hospitality at the European University Institute in Florence where I was able to conduct my work in an inspiring atmosphere surrounded by the beautiful Tuscan hills.

Chapter 1

From Algorithms towards an integrated Framework

Let your workings remain a mystery.

Just show people the results.

(Lao-Tse)

1.1 Problem Description

Modern econometrics relies heavily on the use of computer software to analyse empirical data, as well as to run simulations to investigate the properties of tests and estimators. Complex mathematical algorithms need to be applied to data that is either randomly sampled or that has been observed as the realization of some stochastic process and that is stored in a file or in some database. Researchers who want to perform a certain type of analysis with up-to-date methods basically have two options. By employing standard software packages for econometric analysis, like Eviews or Oxmetrics, they could use a wide range of methods very effectively. The other option would be to take some programming language for statistics, for example Gauss, Ox, Octave, Matlab, or R, and to write or reuse programs that can do the analysis.

The pros and cons of both approaches are quite obvious. If a standard software is used, there is typically well developed graphical user interface (GUI) support,

and the implemented methods are ready to use. However, if some method is missing that is not provided by the respective vendor, extra programming is needed. Although most standard packages also provide a programming interface, it is then usually more effective to apply one of the well established languages for statistics, because often there is already code available which can be reused. Thus standard software lacks flexibility and the possibility to program extensions easily.

By using a programming language for statistics, one has a lot more flexibility to program algorithms. But this approach requires familiarity with the respective language and the resulting programs are usually script-based. This means that it is less convenient and more troublesome to use these algorithms compared to a software with a GUI for interactive modelling. Often even the programmer herself has problems getting a script running that she has not touched for a while. Furthermore, model building in econometrics is typically a multi-step procedure with a number of different algorithms involved. With a script-based approach combining these procedures can become quite a complex undertaking. It always requires text editing of sometimes lengthy source code. Furthermore, documentation is often quite sloppy, which requires to investigate the algorithms themselves to know exactly how parameters need to be prepared and what the contents of the results are. Another problem is that the authors of these algorithms usually see themselves rather as *Scientists* instead of *Programmers* and they often do not reflect very much about software engineering techniques. The result is that software reuse is often limited to reusing single procedures written in some script language for statistics. More complex interactions or object-oriented design is only applied by experienced developers and can still not be considered a mainstream technique in that area.

One of the central contributions of the proposed software architecture JStatCom is that it can be used to increase software reuse, because it provides configurable standard components for recurring tasks as well as mechanisms to use code that has been written already in special languages for statistics.¹ By applying that approach one can develop reliable, feature-rich applications with relatively little effort. More generally, this was one of the major goals of object-oriented programming, but it needs domain specific application frameworks to bring this idea

¹The URL for the framework JStatCom is www.jstatcom.com.

to live. JStatCom is such a framework for data based analysis, especially time series econometrics.

To summarize, the big disadvantage of using special purpose languages to program algorithms for statistics and econometrics is that it often requires special knowledge to reuse them. It is not a solution that can be applied by empirical researchers easily because it often involves time consuming programming or at least adjustments in the source code. This leads to a situation where methods are not being used because they are not part of a standard software and programming is not an option due to resource or knowledge constraints. However, these methods may have been programmed and might already be part of some software library. It would therefore be good to improve the usability of these algorithms by providing a relatively simple way to create user-friendly interfaces for them.

1.2 JStatCom and JMulTi

The aim of this work is to present an approach to creating software for that purpose which is based on the first version of the application framework JStatCom. This software was developed by the author and is based on previous work by Benkwitz (2002), who implemented an early version of JStatCom which was capable to communicate with the Gauss software only. He set up the concept of having a separate Java interface together with a communications interface to an external *execution engine*, a term borrowed from the MMM project (Günther et al. (1997)).

In its current version the framework has been generalized to provide the capabilities to work with a number of computational engines with very different features. It is argued that by now any relevant engine can be used if a communications interface for external procedure calls is provided by the vendor or by other parties. The system has been further generalized to represent new data types, like dates and sample ranges. Many build-in variation points allow to adjust the framework to different modelling situations. For this, a new abstract engine system, as well as a different internal data management had to be implemented for JStatCom, which also affected many components for GUI design. Therefore the whole system was rewritten to cope with the more general requirements. Furthermore, all classes and methods have been thoroughly documented and a complete API spec-

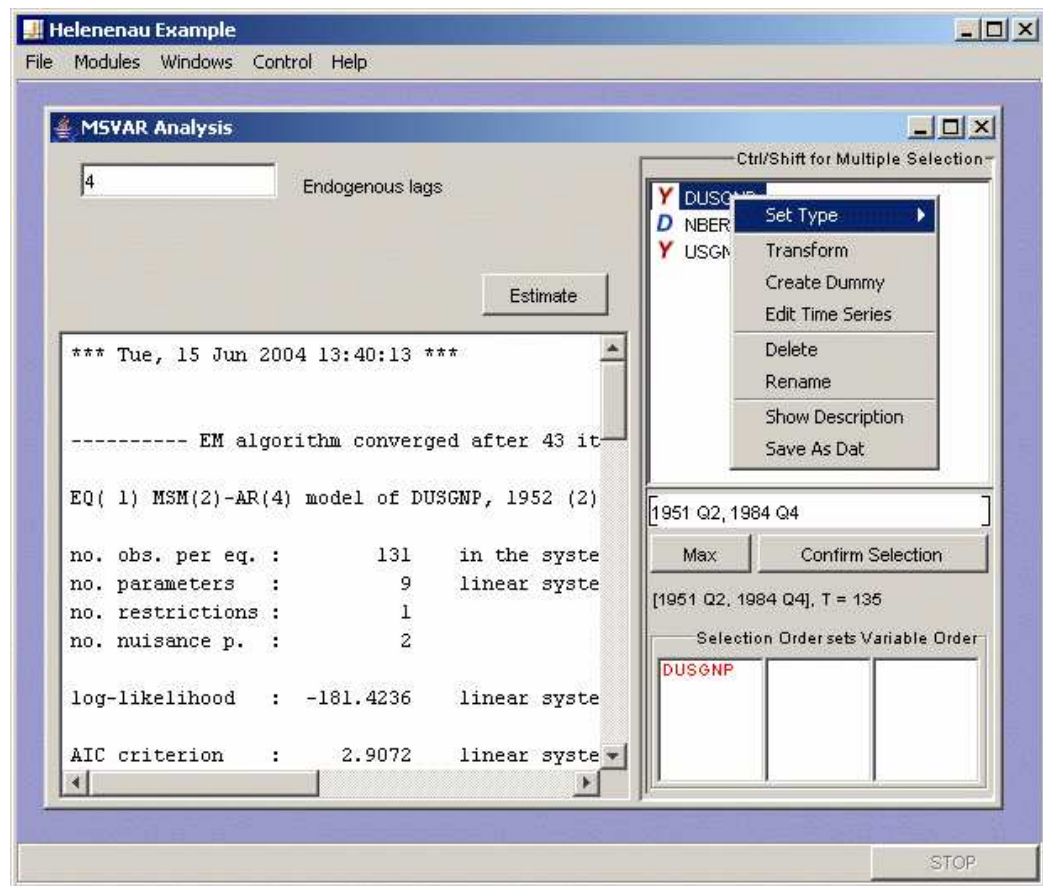


Figure 1.1: A screenshot of a simple example module for Markov-Switching models

ification has been generated which can be used by developers. It should also be mentioned that regression testing has been applied for all features of JStatCom that can automatically be checked via assertions.

On top of it the software JMulTi was developed which is an application that is based on JStatCom.² It is also described already in Benkwitz (2002), but since then many new features have been added. Of course it had to be adjusted in many ways because the underlying framework has been changed. Chapter 4 describes the current state of JMulTi. The author has developed the modules *Initial Analysis*, *STR*, *ARCH*, and *Nonparametric Analysis*. But due to the change in the underlying

²The URL for the application JMulTi is www.jmulti.de.

structure, also the *VAR* and *VEC* modules have been almost completely rewritten. Furthermore, various new features have been added to these modules compared to the version in 2002. A help system was also integrated which is based on the JavaHelp system. It has been generated with the tool *JHelpDev* which has been developed by the author, see Appendix B. In general the software JMulTi is now much more user-friendly, has many more econometric features, and is more reliable. It is slowly getting accepted by the econometric community. The web-statistics for the release 3.11 08/24/04 state that the software has been downloaded about 1800 times in less than 4 months. On average the homepage of JMulTi is visited by about 40 different users every day. It is reportedly being used for teaching as well as for empirical research.

Because major parts of this thesis present the software framework JStatCom, it mainly addresses developers who intend to program graphical user interfaces for mathematical algorithms. The work investigates the requirements for such a software and describes solutions to the typical problems that appear in that context. It tries to adopt to the situation where *Scientists* develop algorithms from their domain specific knowledge and work closely together with *Programmers* who apply the framework to program GUIs for those procedures. Both roles can also be filled by a single person. The latter scenario is not uncommon, examples are Harald Uhlig's *Toolkit* (Uhlig (1999)) or James Davidson's *Time Series Modelling* software.³ However, JStatCom makes it possible that scientists develop their algorithms in a way they are familiar with, and a Java programmer can focus on designing a GUI for them. This way the development process can become more efficient because all participants may use their specific expertise. It is not required that a scientist needs to program in Java and learn about application development. On the other hand the programmer does not need to have a deep domain specific knowledge to implement the GUI with JStatCom.

Figure 1.1 shows a screenshot of a relatively simple module that has been programmed with this software.⁴ Without describing any details in this place,

³The URL of the software is <http://www.timeseriesmodelling.com>.

⁴The module estimates a Markov-Switching AR model with a variable mean and two regimes. The transition probabilities are assumed to be constant, as well as the variances and the autoregressive parameters. It is possible to reproduce the results presented in Hamilton (1989) with this code. The GUI for this provides an interface to load the data, set the sample, specify the number

one can see that there are GUI components for specifying the input for some algorithm. The output of the underlying computation is presented in a text area. It will be shown how modules of similar type can be created in a straightforward way in Java with the help of JStatCom. Domain specific algorithms might be reused even if they are written in another language. The language that has been used in the example for the screenshot was Ox.

Although there already exist a number of solutions to this task, the strength of the presented approach lies in its flexibility and the high level of code reuse that can be achieved. It also promotes an object-oriented design that allows to create scalable applications that do not tend to become much more complicated and error-prone as more features are added, and thus reducing the entropy problems (Bianchi et al. (2001)).

A very general description of the problems that occur when developing software for scientific computing is given by Morven Gentleman in Boisvert and Tang (2001, preface). The author mentions that often complex software systems are created by scientists rather than software engineers. This can lead to the common situation that best practices in software engineering are ignored or not recognized, and that projects can suffer from this deficiency. The lifetime of scientific applications is often measured in decades, although the code is under constant change. Therefore it might well pay off to introduce some modern programming concepts, like for example unit and regression testing, into scientific software development.

An observation that can be made in areas that heavily depend on the use of complex mathematical algorithms is that large and powerful libraries for math, statistics and graphics are created in different programming languages, but that there is a lack of an integrating framework that seeks to make those procedures accessible in a user-friendly way. So far there are only isolated solutions for certain problems, as for example described in Ashworth et al. (2003), but no attempt has been made to standardize the creation of GUI's for mathematical applications in a more general context. One exception is the web based approach MMM which was developed as an architecture to share algorithms and computing resources via the internet. User interaction was done via a browser interface.

of AR lags and estimate the model.

However, this solution was still not convenient enough for users and was therefore not widely used for empirical analysis. The presented software framework JStatCom tries to fill this gap by defining classes that are especially designed to link between existing math libraries and a graphical user interface. It is not focussed on new algorithms for math and statistics, but concentrates on convenient user interface components, an efficient variable bookkeeping system and on a powerful and extendable data model. The main target of the software is the desktop computer, instead of a distributed computing environment, although it is conceptually not limited to a local environment. A special feature of JStatCom is that existing code from popular matrix oriented languages can easily be reused without even changing it. The software makes every attempt to be both, developer- and user-friendly. This is mainly achieved by providing standardized ways to develop and test applications based on it. JStatCom suggests a certain class design that can be applied to many different modelling situations. This way the developer can always apply a very similar class structure, although the underlying models might be quite different. How this can work in practise is described in Chapter 4.

It has to be mentioned that a framework can never be developed successfully without using it in a real world application, see for example Venners (2002). Hence, JStatCom contains all general solutions that have been found when developing JMulti, the first application that is based on it.

1.3 Access to Algorithms

This section describes which options exist to use JStatCom together with an external execution engine, a Java library, or a *native* library to get access to already implemented algorithms for a given problem.⁵

1.3.1 Using an Execution Engine

The framework in its current state does not depend on a special engine anymore, like for example GAUSS, but can communicate with a number of different programs. In principle it is now possible to integrate any kind of execution engine

⁵The term *native* is used to describe operating system dependent binary resources.

that has a programming interface for calling procedures provided by that engine. Such interfaces exist for many packages used for scientific and engineering programming, such as

- GAUSS (Vinod (2000))
- MatLab (Cribari-Neto and Jensen (1997))
- Ox (Cribari-Neto and Zarkos (2003)), free for academic use
- Mathematica (Rose and Smith (2002))
- R (Cribari-Neto and Zarkos (1999)), GNU open-source project
- Maple (Hutton (1995))
- Xplore (Härdle et al. (1995))
- Octave (Eddelbuettel (2000)), GNU open-source project

The references link to articles describing how the respective product is used for statistics and econometrics. Most packages are published under a commercial license, but there is also a number of high quality open-source tools available, of which especially the R project is becoming increasingly popular.⁶

The use of an external engine has the advantage that most problems related to numerical accuracy, performance and the availability of efficient algorithms and libraries of high level functions can be delegated to an already existing and well established package. In addition to that, users of that package are already familiar with the respective scripting language and can use it efficiently to implement their algorithms. JStatCom does not make the attempt to compete with any of those products, but aims at enhancing their value by providing support for the creation of very flexible GUI's. This way it can be used to create stand-alone applications that could even incorporate code from different packages, say, GAUSS, MatLab, and Ox, just to name some of the most popular software tools used in econometrics. The disadvantage of using a separate program for the numerical computations is that the portability of the resulting application is reduced, compared to that of a

⁶The URL of the R project is *www.r-project.org*.

100% pure Java program. This is because the communication between the Java side and the external engine can only be done efficiently by using operating system specific functions, also called native methods, thus limiting one of the greatest advantages of the Java language. However, this limitation can in principle be overcome by providing communications schemes for different operating systems, if the respective engine allows this.

1.3.2 Using Libraries and Toolkits

Apart from an external execution engine, there is also an increasing amount of programming libraries and tool sets that can be used for various numerical calculations. As the Java programming language is gradually gaining acceptance also for high performance numerical computing, see for example Boisvert et al. (2001) for a discussion and Bull et al. (2001) for benchmark comparisons, there are by now various Java packages available that can be used for statistics and econometrics. Problems that still remain when developing numerical libraries for Java are summarized and evaluated in Boisvert et al. (1998), most of them are still valid today. Nevertheless, the authors give an optimistic outlook that a combination of new language features and optimized compiler performance will provide further improvements. Among the needed enhancements are support for complex numbers with the same performance as for primitive types (Wu et al. (1999)) and a multidimensional array package (Moreira et al. (2001)). These issues are addressed by the Java Grande forum and have led to a number of proposals for changes in the language specification as well as new extensions for numeric libraries.⁷ By now there already exist a number of libraries relevant for statistical computing that can readily be used.

Among the most comprehensive linear algebra packages for Java are:⁸

- COLT, a toolkit for high performance computing

⁷The URL of the Java Grande forum is www.javagrande.org.

⁸The links to the homepages of the mentioned projects can be found at math.nist.gov/javanumerics.

- JAMA, a Java matrix toolkit that was proposed as a standard implementation for a matrix package by NIST ⁹ and MathWorks ¹⁰
- JLAPACK, translates LAPACK Fortran routines to Java (Anderson et al. (1999), Doolin and Dongarra (1997))
- JADE, various additions to the default Java environment, including linear algebra classes
- Java Numerical Toolkit, preliminary proposal for a numerical library by the Java Numerics Working Group
- JMSL - numerical library for Java, a suite with a large collection of math and statistics functions under a commercial license.

For JStatCom these developments have been encouraging, because using calls to a Java numerical library would be the most natural way to incorporate complex math algorithms to a Java application. As opposed to using a native execution engine, this does not affect portability, meaning that applications could be run on all platforms for which a JRE exists. In addition, deployment would be much easier, because no extra resources from other languages must be shipped, like for example Ox compiled classes or GAUSS gcg files.

Apart from employing an external execution engine or a Java numerical library, JStatCom also provides support for calling native libraries directly without the need to write a dedicated Java wrapper for each function via the JNI interface (Liang (1999)), see Section 3.17. This opens the door to use existing and well established libraries for numerical computing, originally written in C or FORTRAN for high performance calculations, like for example the numerical recipes toolkit (Press et al. (2002)). However, using this scheme again leads to the portability problem mentioned earlier, because of the necessary native calls. Apart from that, it introduces pointer manipulations to the Java implementation. Pointers are direct references to memory locations through which the contents of the virtual memory of a running application can be accessed and modified without further checks.

⁹The National Institute of Standards and Technology, *www.nist.gov*.

¹⁰The URL is *www.mathworks.com*.

Pointer manipulations and pointer arithmetic (Deitel and Deitel (2002, chap. 5)) can offer great performance gains, but are often the cause of subtle programming errors. Although in JStatCom the mentioned pointer operations are almost completely hidden behind a standard interface, it can make applications less stable and exposes them to the possibility of sudden crashes, if an error in one of the native calls happens. Therefore another advantage of the Java language, automatic array bounds checking and the absence of memory protection errors, is given up for the sake of greater flexibility if this type of method call is used.

How the algorithms for the actual computations are implemented is left to the developer. The decision will mostly be based on what resources are available to solve a concrete problem. In the case of JMulTi, there existed a large code base of GAUSS libraries for various econometric methods, which was the reason to develop the respective interface first. In the meantime, there were requests to implement methods written in other languages as well. This has led to the described generalizations in the communications scheme as well as in the data structure.

1.3.3 Including Graphics

So far the possibilities of implementing numerical calculations have been considered. One very important aspect of data based analysis is the graphical representation of data and results in a quality that allows it to be used for publications. There are again several options that are available for JStatCom which focus on reusing existing solutions. The creation of a new graphics engine would be beyond the scope of this project and is not considered as an option.

First, if an execution engine is used, then the respective language usually supports the creation of graphics of various types. In general this functionality can be used in the same way as the numerical functions. However, as graphics calls often require extra packages, create temporary files, and can be configured in various ways, there are sometimes small deviations from the calling procedure without graphics. These differences depend on the respective engine and will be explained in the next chapter for each case. In JMulTi the graphics features of the GAUSS execution engine have been used.

A second option to create graphics from within JStatCom is to use a special graphics engine for that purpose. One possibility would be the program Gnuplot.¹¹ It can create various kinds of 2D and even 3D plots and is currently used by the Octave statistical program for displaying graphics. It is also mentioned in Cribari-Neto and Zarkos (2003) as a possible alternative to the Ox graphics engine. Gnuplot is open-source and runs on all major operating systems. There exist several programming interfaces to enable calls from other applications to Gnuplot. These features make it a reasonable candidate for a statistical graphics engine. Currently Gnuplot is not used directly by JStatCom, because it would require a specific communications interface that works with temporary files. But it can be employed by the Ox program to display graphics with the Ox package `gnudraw`.¹² In this case the Ox communications interface as described in Section 3.16 could be used.

Like for numerical libraries, the best solution would be to have a 100% pure Java implementation of a graphics engine. This would keep portability intact and it would simplify deployment. One of the most promising solutions is `jfreechart`.¹³ It is again an open-source project that aims at the development of a feature-rich graphics engine for Java programs. Except 3D surface plots, it can create most kinds of graphs that are relevant for statistical analysis, also with the possibility to configure the display interactively. Out of these reasons, `jfreechart` would be another good option for displaying graphics from within JStatCom. This library is not integrated in the framework yet, but can of course be used in the Java code without special support by JStatCom. However, it would be helpful to provide adapter classes that simplify the creation of standard plots and that work well with the internal data management system of the framework. These possibilities will be further investigated in the future.

¹¹The URL of the gnuplot project is www.gnuplot.info.

¹²Charles Bos has developed this package which is available from his homepage www.tinbergen.nl/~cbos.

¹³The URL of the jfreechart project is www.jfree.org.

1.3.4 Integrating available Tools

It was shown that for all special tasks related to data based analysis there already exist tools that can be reused. What is missing is a framework that can be used to combine the libraries and toolkits of choice efficiently. JStatCom tries to do just this. By having the possibility to use many different computational engines and/or numerical libraries, the framework is usable for a wide range of applications.

Another feature that is provided by JStatCom, is support for creating helpsets for user interface components. As a help system should give users an idea of how the underlying procedures operate, a high degree of formalization is often necessary for the help text, for example to describe the theory of estimating a Smooth Transition Regression Model (Teräsvirta (2004)). Current help authoring systems lack explicit support for math typing. It seemed logical to rely on the well established standard solution LaTeX that many researchers are already familiar with. Together with latex2HTML, a software to convert TeX documents to the HTML format, and the developed tool JHelpDev it is now possible to reuse LaTeX documents to create a state-of-the-art JavaHelp system (Sun Microsystems (2003)) efficiently, see Appendix B.¹⁴

As JStatCom and JMulTi have been developed to be used for econometric analysis, especially multiple time series analysis, JStatCom contains data types and user interface components that are useful in that context. However, the framework is not restricted to be used for time series analysis exclusively, but can be augmented and customized for various environments. A strong emphasis of this thesis is to show all directions in which JStatCom can be extended without changing the current class structure and without breaking binary compatibility with previous editions.

It was an important goal to make the framework accessible to users who do have some knowledge in programming, but little or no background in software engineering. This means that it should be easy to create simple applications. On the other hand, JStatCom should also support the creation of larger, more complex programs. It is hoped that the proposed set of solutions is simple yet flexible enough to meet these competing requirements.

¹⁴The URL for the help authoring tool JHelpDev is jhelpdev.sourceforge.net.

1.4 JStatCom in comparison to other Approaches

As empirical research in econometrics can only be done with specialized software packages, this project is not the first attempt to provide methods with an intuitive GUI. Other approaches to create applications for statistics and econometrics exist. The following section describes some of these solutions and relates them to JStatCom.

1.4.1 MulTi

One of the earliest projects that adopted this approach was MulTi 1.0 (Haase et al. (1992)). It already employed GAUSS as a computing engine but used special language features to implement a graphical user interface on top of it. The program had a rich set of features and even implemented VARMA modelling capabilities (Lütkepohl (1991, chap. 6)) that are still unique among econometric packages. With the introduction of GAUSS for Windows, the support for GUI commands ceased, and thus rendering further development of MulTi 1.0 impossible. However, there were also limitations in the design of the software, letting the cost curve of additional changes and maintenance become very steep. Therefore the project, although very successful at that time, was cancelled.

1.4.2 Observations on Entropy and Software Reuse

An observation from the experiences with MulTi 1.0 and similar projects is that entropy limits the lifetime of such software by making it literally impossible to maintain and extend an existing system with a reasonable effort after it has grown beyond a certain size. If important project members leave the development process, this often means the end of lifetime. Benkwitz (2002) identified the reasons for this phenomenon in the context of econometric software by the lack of object-oriented engineering principles, and, in case of GUI based applications, the mangling of GUI and algorithm code. The latter is often the simplest and fastest way to create working prototypes, but it is only suitable for small projects with a limited number of features. He promotes a clear separation between the GUI and math code, as already argued in Liu et al. (1995). This is in line with well

established software engineering standards. An abstract model for that programming paradigm is the Model-View-Controller architectural pattern that promotes the separation of data, rendering of data and the translation of user interactions into model changes (Eckstein et al. (1998), Sun Microsystems (2002)).

There are a number of other problems that increase entropy of software projects. One is the lack of documentation which makes it often hard for new project members to understand an existing system. Maybe even more severe is accumulated complexity and cross-references between various components of the system that are difficult to trace back. In a procedural programming context this means that there are large monolithic procedures, as well as many global variables that might be modified in different parts of the program. Making the step towards object-oriented programming does not necessarily mean that those problems will be eliminated. In fact, they may become even worse, because class structures can become extremely hard to understand. This may happen due to bad initial design and due to a lack of refactoring during the development process (Fowler (1999)). Although an object-oriented approach is often conceptually better than using procedural programming techniques, it usually requires a deeper understanding of how to design, implement and extend such a system properly. It uses a number of abstract concepts that many researchers are not familiar with. This leads to a common dilemma with code written for scientific applications: there exist many good algorithm implementations, but there is a lack of structure and therefore limited reuse. One of the promises made by the object-oriented programming paradigm was that there would be an increase of software reuse. However, reusability of object-oriented systems is often limited, see for example Pree (1997). It is not a natural property of such a system, to quote Gamma et al. (1995):

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder.

It takes a lot of experience to design a reusable object-oriented system. A reusable class can almost never be designed from scratch, but is the result of a stepwise refinement process. In that respect, a simple procedural programming approach often gives better results in terms of reusability, if only more or less isolated algorithms are considered. This is, how reuse works best in scientific

computing, because the conceptional overhead of understanding the relationships between many classes is usually greater, than just calling a procedure of a library that takes a number of parameters and returns a specified result. To summarize, designing reusable object-oriented systems is hard and too time consuming for many researchers. Designing reusable algorithms is much easier, because they are often closer to the real problem and do not require many software specific abstractions. But especially for creating applications with GUI components, a procedural approach leads to the mentioned entropy problems. However, GUI components can increase the usability of many procedures and are sometimes even necessary, because of the complexity of the underlying procedures.

The proposed solution is to split the development of algorithms and GUI code and to keep the two strictly separated. This way, algorithms do not necessarily have to be object-oriented, but the GUI system would be, thus being extendable and maintainable also for larger projects. The software providing the GUI would preferably be a framework which is defined as a set of reusable classes that make up a reusable design for a class of software (Johnson and Foote (1988), Deutsch (1989)). That means that it already provides a structure as well as key functionality for applications in a certain problem domain. The designer of an application can reuse not only classes, but the whole design of the framework and concentrate on specific aspects of his implementation.

In the following, other, more recent approaches than Multi 1.0 will be described, focussing on how the different solutions achieve the goal of creating an extendable software system.

1.4.3 The MMM Project

Although the already mentioned MMM project has meanwhile been cancelled, it bore some similarities to JStatCom and is interesting to look at. It was intended as a general interface to various different execution engines that can run methods for statistical computations. These methods could be provided by statisticians, who use their favourite programming environment to create them, thus adopting to the habits of algorithm reuse. The goal was to provide a general interface that allows to combine a growing number of statistical procedures easily. All services

were provided via the World Wide Web in a distributed environment. The user could access these services via a browser interface. The use of Java to create more sophisticated user interfaces was considered as well. Therefore the goals of MMM were comparable to those of JStatCom, although a great focus was on platform independence and the availability of the service via the Internet. The reason was that MMM should eventually be used as a platform for component leasing on the Web (Riessen et al. (2000)). As compared to JStatCom, similar problems had to be solved:

- communication to different engines via a unique interface
- providing each engine with the necessary libraries and configuration information needed to run properly
- transforming data to a format understandable by the respective execution engine
- providing the capability to combine arbitrary methods
- saving intermediate results, maintaining the state of the analysis
- making it easy to check in new methods

The solution to all these problems is not trivial in a local environment, but is truly complex in a distributed environment where potentially many requests have to be dealt with at the same time. Security issues are a severe problem, because users can basically control the execution of methods on a server which is a potential danger, if security holes in an engine implementation can be exploited. Apart from the purely technical challenges, it seems at least questionable whether it is possible to achieve a satisfactory standard of the user interface. The reason was that the MMM approach introduced a great deal of flexibility, but at the same time it put the burden of how to combine different methods in a meaningful way on the user. Just providing a way to call different algorithms is certainly not enough, because there is a lack of methodology. Furthermore, it imposes restrictions on the interfaces of the statistical procedures that have to be combined. For example, some statistical test routine might take residuals as a matrix with dimensions

$T \times K$, but the output from the estimation routine might be $K \times T$, where T denotes the number of observations in a sample and K the number of variables in a regression. Sometimes lag truncated series are assumed for an estimation routine, otherwise the raw data. These small differences can cause a major headache and can significantly decrease the acceptance of such a system. Taking the experiences with MMM into account, JStatCom makes fewer promises, but tries to adopt to the requests of potential users who need a user-friendly software for applying advanced numerical methods. Like with the MMM approach, it is possible to combine different engines and arbitrary algorithms. However, the difference is that the set of available methods and the order, in which these methods can be executed, must be defined by the programmer. The user should only be given the choice between procedures that are meaningful in a certain context. Given that the approach taken with JStatCom works well in a non-distributed environment, one could think about ways to combine it with network services, if there is any need for such a feature.

1.4.4 The OmegaHat Project

An interesting open-source project to develop statistical software is OmegaHat.¹⁵ It was started in 1998. The project provides a language that is very similar to the Java programming language, but that can be used interactively. This means that expressions can be evaluated and executed directly on a command prompt. Most statistical programming environments are interactive. The special feature of OmegaHat is that the language can serve as an integrating framework, or an umbrella system that uses various different tools via a single language interface. It provides interfaces to the statistical programming languages R, S and S-plus. OmegaHat is therefore not just another statistical language, but a programming environment with similar goals as JStatCom. However, the latter has a clear focus on GUI interface components and provides a reusable design for statistical applications. The OmegaHat language could be used by JStatCom to access an execution engine, thus serving as a layer between the GUI interface and the execution engine. Therefore it might be a great resource for various tools that can

¹⁵The URL of the OmegaHat project is www.omegahat.org.

be used with JStatCom, because one of the goals of the OmegaHat project is to develop Java packages for statistical computing.

1.4.5 Creating GUI's with Matlab

Matlab is one of the leading commercial solutions for numerical calculations in statistics and engineering. The language can be used not only for programming algorithms and calling graphics, but also to create graphical user interface components. Together with the Matlab Compiler, it is possible to create stand-alone applications with a graphical user interface. In fact, this approach is used for many useful tools used in statistics and economics, for example Uhlig (1999). Although the solution is often very convenient and serves the needs of many researchers, its capabilities to create and maintain more complex GUIs are restricted. Matlab provides the development tool GUIDE for the purpose of designing user interfaces. It is comparable to visual builder tools for Java or other languages. However, the system provides far less components and is less flexible as compared to Java Swing, for example. Although simple applications can be created quickly, increasingly complex projects tend to be hard to extend and maintain because the GUI components are not defined as objects or components. Therefore it is not easily possible to split a complex GUI into several smaller components that can then be plugged together. Instead, the approach tends to create large monolithic files where various callback methods are defined, which are invoked from the GUI controls. There is no straightforward way to disentangle a complex application into smaller components that are relatively independent of each other. As a consequence, this can lead to the cancellation of otherwise successful projects or it can limit the potential to extend an already existing modelling toolkit. Another result of using this scheme is that Matlab applications tend to be dialog driven, because this is a way to combine several simpler GUIs in a single program to achieve the desired functionality. The reason is that there are no containers available that can hold many panels simultaneously which can be accessed via menu items, for example. Instead, new dialogs appear which hold the GUIs for different parts of a program. This may result in an unsatisfactory user experience as it clearly limits the possibilities for designing a GUI according to current standards.

JStatCom could be an alternative solution to create stand-alone programs, using procedures originally coded in Matlab and compiled with the Matlab Compiler. In this case, no GUI programming with Matlab would be necessary. Projects using this approach can expect to be extendable and maintainable, even if many features are implemented, given that some general design rules are followed. There are no limitations in the programming of the user interface other than those that the Java language imposes. Matlab routines could even be combined with code from other languages. The drawback is that some Java programming would be necessary, instead of using only Matlab functions. For very simple applications, this effort would not pay off, but for simple GUI's there are no problems with Matlab either.

There are also other statistical packages that provide language features to create graphical user interfaces, for example Xplore with so called Quantlets. The main critics of these systems is always the same, they are great to create simple dialogs or wizards, but they are not optimal for the creation of more complex interfaces, or even applications with lots of features. JMulti is an example for a software that could not have been created with only the help of the available tools.

1.4.6 Creating interactive Programs with Ox

Ox is a matrix oriented programming languages that introduces object-orientation to algorithm coding (Doornik (2002a)). In that respect it differs from other solutions and provides the potential to create even reusable object-oriented systems. The software provides an API that makes it possible for external applications to call predefined procedures and to create objects. This feature is exploited by JStatCom, and it is therefore possible to use Ox code together with the GUI building capacities of the framework.

An extension of Ox is OxPack (Doornik and Ooms (2001)). Together with GiveWin, a graphical front-end that provides general functionality for all GUI modules, it can be used to create graphical interfaces to a model. The difference to the previously described approaches is that here an object-oriented approach was taken to provide GUI functionality. It is necessary to subclass the `ModelBase` class which is then used by OxPack to set up the display of the user interface for

the created model. Figure 1.2 shows the relationship of the relevant classes for a hypothetical Smooth Transition Regression (STR) modelling class in a UML diagram. For clarity, the representation of those classes is simplified, not all public methods are shown. The UML notation is a widely accepted standard to describe software systems, see Booch et al. (1999) for an exhaustive discussion.

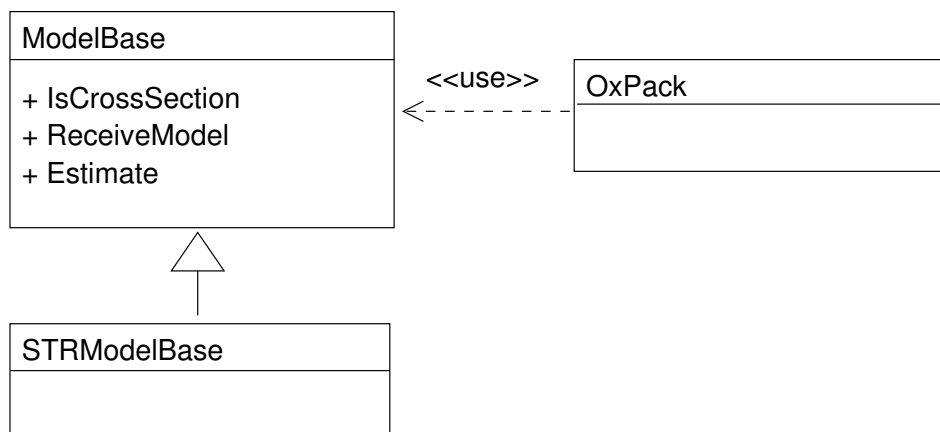


Figure 1.2: Class diagram for an interactive Ox program

Subclassing means that all functionality from a superclass is inherited, but that behaviour can be redefined by providing different implementations for certain procedures. The signature of these procedures does not change by overwriting them. A subclass can always be used instead of a superclass, because it *is* an instance of that class. Therefore `OxPack` can take the inherited class `STRModelBase` as an argument to set up the user interface according to the definitions laid out in that class. These definitions describe what kinds of user interface components are used, which estimation routines are possible, the name of the model and various other settings. Once understood, this approach can be used to create user interfaces to different models in a fairly standardized way. It even provides the option to define HTML helpsets, a feature that is also implemented for JStatCom modules.

By applying this way of creating user interfaces for econometric models, it is easy to separate algorithms and GUI related code, because the `ModelBase` class is only used to define which algorithms are called according to the user specification.

The actual code for the econometric procedures should be defined in different classes that are independent of the interface definition and that could even be used by other user-defined models.

There is only one problem with this approach. Between `ModelBase` and its subclasses must exist a *is-a* relationship. This means, that every new model must be a special case of the general model allowed for in `ModelBase`. The `ModelBase` class is therefore designed to be a generalization of all potential models used in econometrics. It defines so called *virtual* methods that can be overridden by its subclasses. All of these methods are executed by the `OxPack` class in a predefined order which always starts with the model formulation and ends with the model estimation. It is also possible to define the steps that should be carried out afterwards, among which are typically diagnostic tests and the graphical analysis of the estimated model.

Although this scheme allows to incorporate many econometric models and can simplify development greatly, it is also obvious that it restricts the applicability of the design to compatible modelling situations only. Models that require a different GUI behaviour or that belong to a different problem domain might not fit into that framework. Apart from that, the behaviour of the user interfaces that can be created is pretty much predetermined by the `OxPack` class. Following the definition in Gamma et al. (1995) the used design pattern is a *Template Method*. A consequence of using this pattern is that the sequence of calls cannot be altered, but only the behaviour of the single steps. This means that the flexibility of this approach to create interactive GUI's for various different models is limited. For example, there is no way to incorporate special GUI components to further simplify certain tasks. One can think of a clickable matrix display for selecting restrictions for tests or estimation. There is also no way to set up an event-based communication between different components such that the behaviour of the program is adjusted dynamically according to the selected model or the available data.

The more general problem behind this is discussed in Bloch (2001, item 15). Inheritance is a powerful concept, but it creates static relationships between classes and should be used only, when a true *is-a* relationship exists between the superclass and its subclasses. An alternative concept that can often replace inheritance constructs is *composition*. Composition means that a class is not an ancestor of an-

other class, but that it keeps just a reference to that class to get access to the needed functionality. Applied to the design used by Ox, this means that limitations stem from the fact, that not every model should be derived from the `ModelBase` class. An alternative would be to use a composition approach, where different classes or components provide the necessary functionality to create a GUI. This scheme could be used by arbitrary model implementations. In fact, this is exactly what JStatCom does. There is much more freedom to design model interfaces, but there is also less predefined structure.

1.5 Concluding Remarks

From the foregoing discussion I would like to draw a number of conclusions. First, there exist many tools that can be used to convey a data based analysis. There are interactive programming environments and there are libraries and toolkits, some of them are implemented in Java. Although a lot of effort is invested in programming new algorithms, it seems that the design of user friendly interfaces lacks behind. Therefore it was natural to invest some effort into developing a general solution that could potentially make use of the available tools, thus avoiding to “reinvent the wheel” by any means.

Second, the comparison to other approaches to create GUI applications for data based analysis showed some deficiencies of current solutions. Either the created application will suffer from entropy, like for example with Matlab based programs, or it will not be flexible enough to allow for the implementation of arbitrary models, like with the Ox based approach. I described, how JStatCom seeks to avoid these problems by applying best practices in current software engineering. How this has been implemented and could in fact be applied for real-world projects will be described in the remaining chapters.

A final remark is on the use of the Java language. Benkwitz (2002) has already given a number of arguments, why this was a reasonable choice. Taking the ongoing developments into account that have led to further performance improvements and a growing number of libraries, also for numerical computing and graphics, his reasoning can only be reconfirmed. Other numerical tools, for example Matlab, also rely on Java when it comes to user interface creation. JStatCom does not only

implement user interface components, but also a complete internal data management system. Lightweight programming languages, like VB, would not have been an appropriate platform to build such a system on. The potential alternative would only have been C++, or the newer C#. C++ itself offers too many features that are a constant source of errors. Furthermore, it does not have automated memory management, imposing the burden of allocating and deallocating memory on the developer. A superior solution to C++ would be Microsoft's C# language. It has some slight advantages over Java, simply because experiences from using Java have been taken into account when it was developed. But Java is not a static language, hence some of the critics are already addressed with the current release J2SE 5.0. Apart from that, a major drawback of using C# is that GUI applications written with this system can only be run on the Windows operating system. True portability of complex graphical applications is a feature that is almost unique to Java. Therefore JStatCom can in principle be used on any operating system that is Java enabled. However, if native libraries are used for engine communication or library calls, these specific features would not be portable.

1.6 How to read this Thesis

In this first chapter, the developed solution was compared to other approaches and is put into a broader context. It was shown that most concepts have been used before in similar situations, but with different results. The potential uses of JStatCom were explored and the framework was related to existing solutions.

The second chapter provides a detailed step-by-step example for how to use the framework to create a GUI module for a statistical test. The described scenario is relatively simple but quite common for econometrics. It should be possible to reproduce the example with only little knowledge in the Java programming language.

In the third chapter, a detailed description of the software architecture will be given. Certain implementation aspects will be explained when necessary. The aim of this chapter is to document JStatCom for potential developers and maintainers. It should be mentioned that reading all of this chapter is not required to start programming with JStatCom. However, developers should get an overview of the

available systems. They could use this text like a reference manual where only the needed parts are read when required. The specification of the Java programming interface of the framework is the API documentation. It can be found on the web under *www.jstatcom.com*. Having this documentation at hand can help to understand the description of certain classes better.

The fourth and last chapter provides examples of what can be done with JStat-Com. It will describe the reference application JMulti from a developer's point of view with a focus on how certain solutions could be achieved with the help of the framework.

The text should be understandable to statisticians and econometricians, therefore technical terms that are familiar to computer scientists are explained when they occur for the first time. Appendix A has a guide to the UML notation that has been used throughout the text. However, a general understanding of basic programming concepts is assumed.

Chapter 2

A Motivating Example

2.1 Introduction

This chapter gives an introductory example on how to use the software framework JStatCom. It should serve as a motivation for the architecture documentation in Chapter 3. The example demonstrates, how a graphical user interface for a statistical test could be implemented. This is a relatively simple but realistic scenario. The underlying algorithm is implemented in the Gauss programming language which is quite common for econometrics. Readers who consider using the software framework for similar purposes are encouraged to follow the described steps and try it out.

The example uses the Eclipse Integrated Development Environment (IDE) which is an open-source project sponsored by IBM. This software is very popular in the Java community, also because it is freely available. But it should be stressed that this is just one tool among others and that the use of JStatCom is by no means related to any specific software development tool. Although not recommended, a simple text editor for Java coding would also be sufficient. Because this example should be reproducible, every step is documented including the setup of a project for the Eclipse IDE. Experienced Java developers can certainly skip those sections. Furthermore, it should be mentioned that the example is done under the Windows XP operating system.

2.2 A Step-by-Step Example

The example will implement a GUI for the univariate ARCH-LM test (Engle (1982)). This test is based on fitting an ARCH(q) model to the estimation residuals \hat{u}_t ,

$$\hat{u}_t^2 = \beta_0 + \beta_1 \hat{u}_{t-1}^2 + \cdots + \beta_q \hat{u}_{t-q}^2 + error_t, \quad (2.1)$$

where the β_i denote fixed coefficients, and checking the null hypothesis

$$H_0 : \beta_1 = \cdots = \beta_q = 0 \quad \text{vs.} \quad H_1 : \beta_1 \neq 0 \text{ or } \dots \text{ or } \beta_q \neq 0.$$

Under normality assumptions the LM test statistic is obtained from the coefficient of determination, R^2 , of the regression (2.1):

$$ARCH_{LM}(q) = TR^2.$$

It has an asymptotic $\chi^2(q)$ distribution if the null hypothesis of no conditional heteroskedasticity holds. There is also an F version of this test that is based on the corresponding Wald statistic.

One can see that the test regression 2.1 is uniquely determined by specifying the residual vector \hat{u}_t together with the lag length q . A GUI for that test should therefore allow the user to select a time series and an integer number greater than zero. The test algorithm should then compute the test regression as well as both test statistics. It should also return the corresponding p-values to enable the user to quickly decide whether the null hypothesis is rejected or not, without the need to look up the critical values in some table. The Gauss code for this is given in Section 2.2.10.

2.2.1 System Requirements for this Example

- Windows 2000/XP
- working Gauss installation version 3.2 or higher

Although the Java part of JStatCom can be run on almost any platform, this demonstration requires an installed Gauss. The communication scheme between

Gauss and Java has only been implemented for the Windows operating system. However, Gauss code could also be executed on other platforms with the help of the Gauss Runtime Engine (GRTE), which is a special Gauss distribution provided by Aptech. The GRTE allows for royalty free redistribution of compiled Gauss code that can be executed without an installed Gauss, thus enabling developers to ship stand-alone applications to their customers. JStatCom distinguishes between Gauss and the GRTE, although the two engines can execute the same Gauss code. But to develop with the GRTE one has to purchase a specific license first, therefore it would not be a very good candidate for this introductory example. At least, it is more likely to have a working Gauss installation available.

It should be mentioned that the Gauss installation is only required to incorporate external code. But this example can also be run without this specific algorithm implementation. In this case the procedure for the test should be implemented in Java, which is also an option. Therefore readers are nevertheless encouraged to follow the remaining steps to get an idea about the workings of JStatCom.

2.2.2 Step 1: Download/Install the Java Software Development Kit (J2SE SDK)

Because Java code needs to be compiled and the created binary files (classes) must be executed with the Java Virtual Machine (JVM), it is necessary to download and install the current J2SE SDK from Sun, the URL is *www.java.sun.com*.¹ This software is also required to run the Eclipse IDE software, because it is written in the Java programming language as well. The installation follows a standard setup routine and should pose no problems.

2.2.3 Step 2: Download/Install the Eclipse IDE and some Plugins

As already mentioned, the Eclipse IDE is just a suggestion for use as a development tool. One of the great strengths of the Java platform is that there is excellent

¹The most recent release is J2SE 5.0. With this version the term SDK (Software Development Kit) changed to JDK (Java Development Kit).

tool support. Therefore it would equally be possible to apply other tools with very similar steps.

First the IDE needs to be downloaded from *www.eclipse.org/downloads*. There are usually many different packages for various purposes available which might be a bit confusing. At the time of this writing the needed package is `eclipse-SDK-3.0.1-win32.zip`.² It is likely that the version number will already have changed when reading this text. The software just needs to be unpacked to some directory and will install itself when it is first started.

Because programming with JStatCom typically involves the layout of GUIs, a visual interface builder can speed up development significantly. Such a tool is not part of the standard Eclipse package, but can be installed as a plug-in. The following additional packages need to be downloaded from *www.eclipse.org/tools*. The names of the relevant archives are given in brackets, although the version numbers are likely to have changed already:

- EMF (`emf-sdo-runtime-2.0.1.zip`)
- GEF (`GEF-SDK-3.0.1.zip`)
- VE (`VE-runtime-1.0.1.1.zip`)

Installing those plug-ins merely requires to unpack the archives to the Eclipse installation directory and to restart the IDE. All mentioned tools can be downloaded free of charge.

2.2.4 Step 3: Download JStatCom and unpack it

The last required download is the JStatCom software framework itself, which can be found under *www.jstatcom.com*. The archive `jstatcom.win-1.0.zip` just needs to be extracted to some folder. By default it creates a directory `jstatcom`. It is recommended to rename this directory to something more descriptive for the respective project. For the current example the name `testproject` is used.

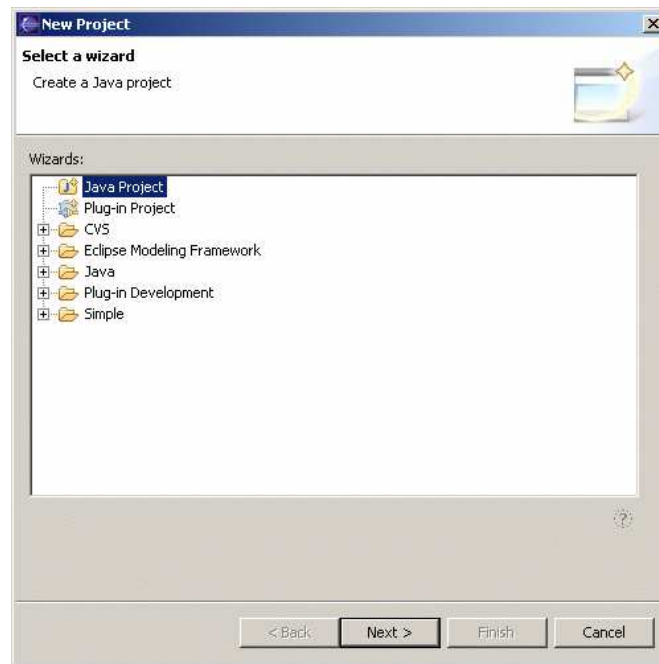


Figure 2.1: Selecting a new Java project in the Eclipse IDE

2.2.5 Step 4: Create an Eclipse Project

In the following it is assumed that JStatCom is located in the directory `c:\testproject`. Now the Eclipse software should be started, the option to change the workspace directory can be ignored. At first startup a help screen is presented, it can be closed. The first step should be to create a new Java project via *File - New*. The resulting dialog is presented in Figure 2.1. When *Next* is pressed Figure 2.2 is shown where the project name should be given, as well as the directory with the resources. The created project can then be seen in the package browser, see Figure 2.3. All jar archives in the project directory have automatically been included in the build class path, for example `jstatcom.jar`, `jama.jar`, `junit.jar`, etc.. By default, all classes that will be compiled go in a subdirectory `bin`. But no classes have been created so far.

²This is the current version as of December 12, 2004.

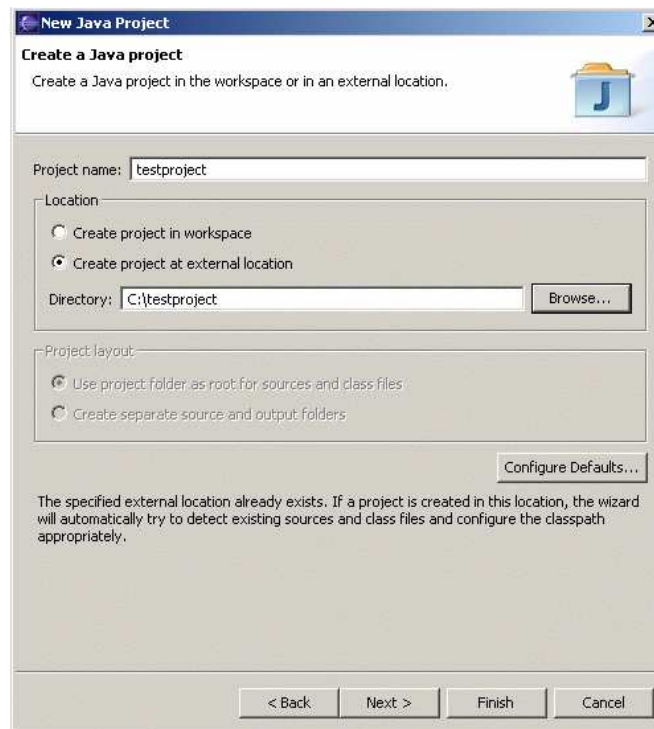


Figure 2.2: Setting project name and directory

2.2.6 Step 5: Create a new Package and a new Class

Programming with Java usually starts with creating a class. The class to create for this example should hold the GUI for the test and should itself be a so called internal frame. This is a special component that is displayed as a window within a desktop application.

Before a new class can be created, the IDE software requires to define a directory with the source code to compile. It will look for Java source files only there and compiles them when changes are made. Figure 2.4 shows the menu that appears on a right mouse click over the project name `testproject`. By selecting *Source Folder* a small dialog will appear asking for the directory name which can freely be chosen, but is typically `src`.

Afterwards the same menu should be used over the just created `src` folder, but this time with the option *Package* to create a new Java package. The dialog for this is shown in Figure 2.5. It is not strictly required to define packages, but it is

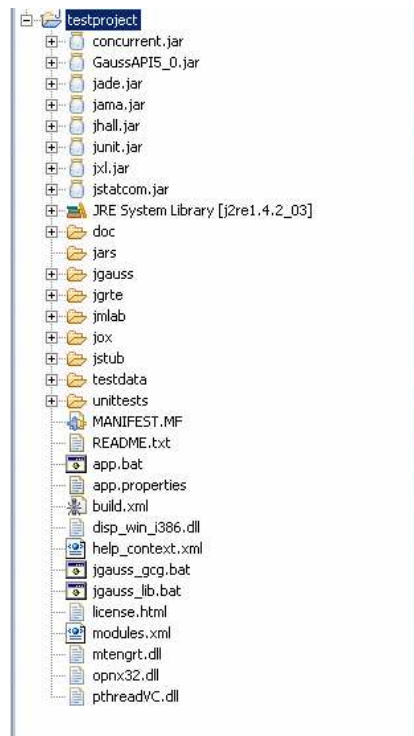


Figure 2.3: Display of the new project `testproject` in the package explorer

good programming practice. The standard for this is the reverse domain name of the organization or firm that develops the Java code. Here `com.myorg` was used as a placeholder. As a starting point one could also use a descriptive name, like `testproject`. It can be changed at a later stage easily.

Now everything is prepared to create the first Java class. Again, the right mouse menu should be used, this time over the newly created package `src/com.myorg` with the *Class* option. Figure 2.6 shows the dialog to specify the class to create. The name of the class can freely be chosen, but class naming conventions for Java suggest that it should consist of nouns, each of them beginning with a capital letter, for example `TestFrame`. As a second step it is important to choose the superclass that the new class inherits from. By default this is just `Object`, the top-level superclass of all classes in Java. However, some more functionality is needed because the class should be a frame holding the GUI for the test. Therefore one should select the class `ModelFrame` with the help of the

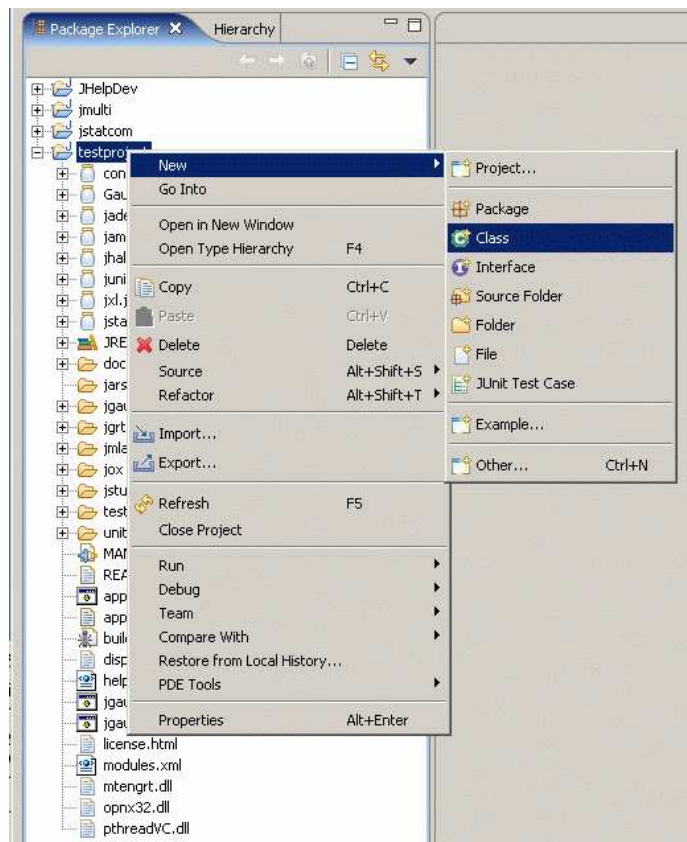


Figure 2.4: Creating a new class

Browse button in the *superclass* field. It is then necessary to remove the default Object entry first and type in the first letters of `ModelFrame`. The dialog will immediately display all possible options. Figure 2.6 shows the result. All other options can be left unchanged.

After finishing the dialog, a Java editor window with the code for the newly created class is shown, see Figure 2.7. Also, the new file is added to the package browser in the source directory `src` under the Java package name `com.myorg`. The class `TestFrame` is a subclass of `ModelFrame`, which is a component from the `JStatCom` system. The generated code is very short, because the new class does not yet define own methods or fields. It inherits all functionality from its superclass. One should consider documenting the header of `TestFrame` with some remarks about the function of this component.

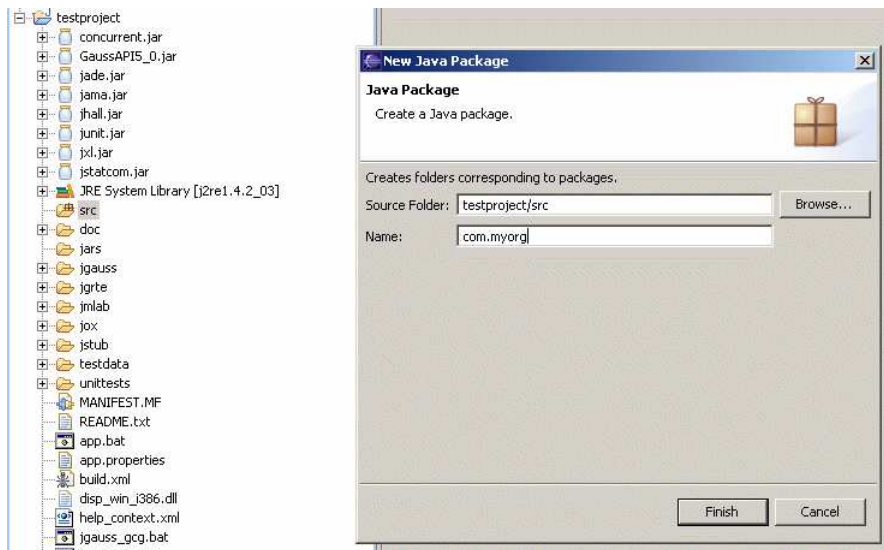


Figure 2.5: Creating a new package

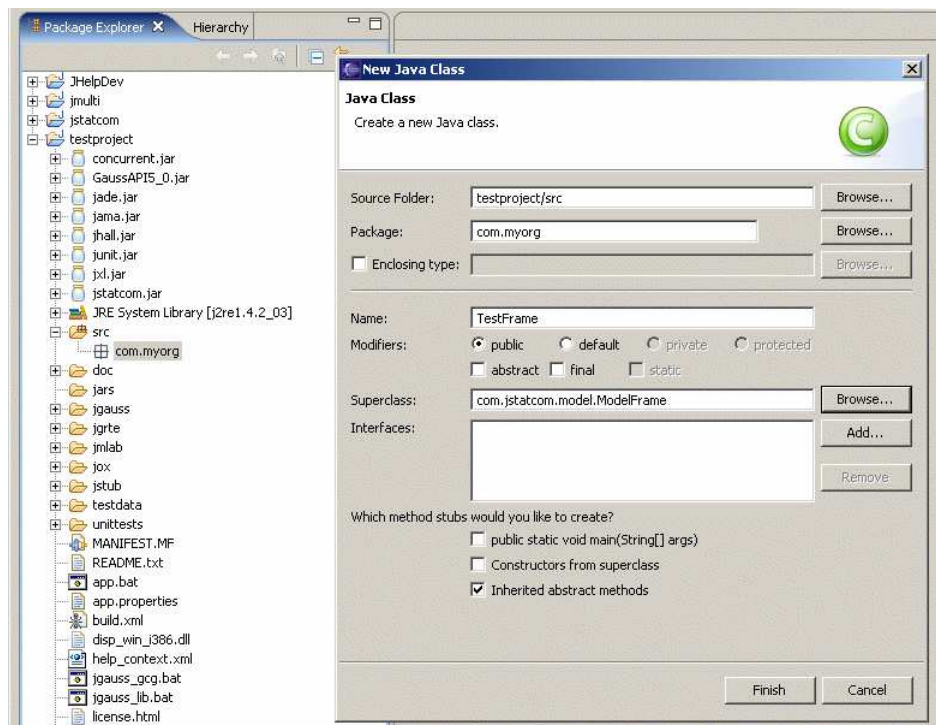


Figure 2.6: Specifying the new class

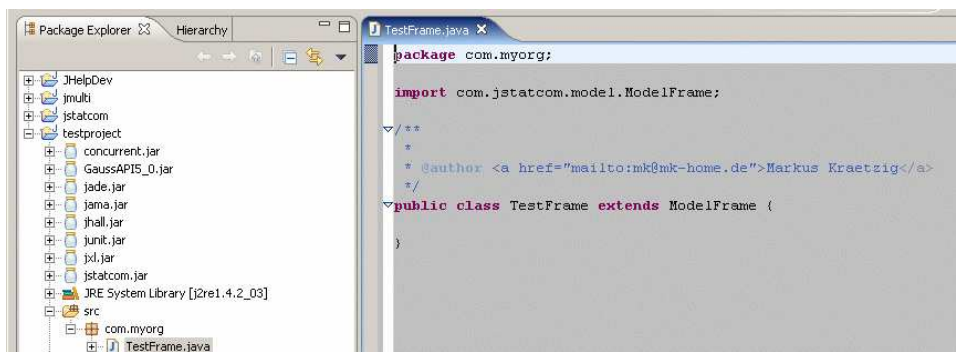


Figure 2.7: Generated Java code for a new class

2.2.7 Step 6: Layout the GUI with the Visual Editor

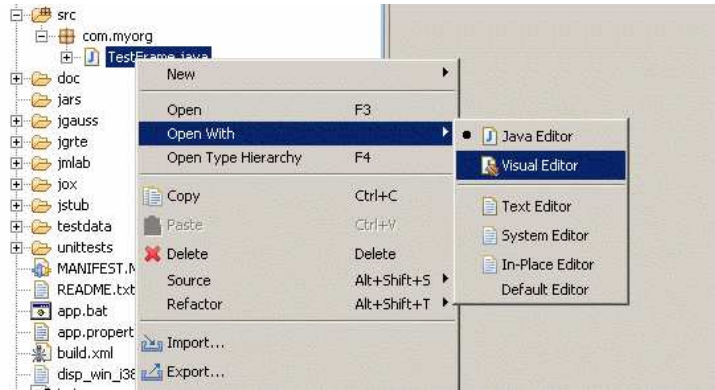


Figure 2.8: Open class in visual builder tool

After having set up the skeleton of the new module, the GUI should be laid out with the help of the visual editor (VE) that has been installed via the mentioned plug-ins.

To use the VE tool in Eclipse, it is first necessary to close the Java editor window that is currently showing `TestFrame.java`. Afterwards the same file should be opened again in the VE by right clicking over the file `TestFrame.java` in the package explorer and selecting *Visual Editor*, see Figure 2.8.

After invoking this operation, it might take a while until the VE is set up, because this is a computationally intense operation. The initial display is shown in Figure 2.9. There is an area at the top where the actual state of the GUI is shown as it would look like during runtime. The frame is displayed with its minimal size in the beginning. The Java editor underneath shows the code that is generated by the VE, which is nothing so far. After all, the VE is just a code generating tool and it is also possible to do manual adjustments by editing the Java code directly. On the right edge there is the so called *Palette* which provides access to many standard Java components that can be picked and placed in the VE. It can also be used to access all components provided by the JStatCom framework, although those are not displayed with an icon but must be chosen via the button *Choose Bean* from the palette.

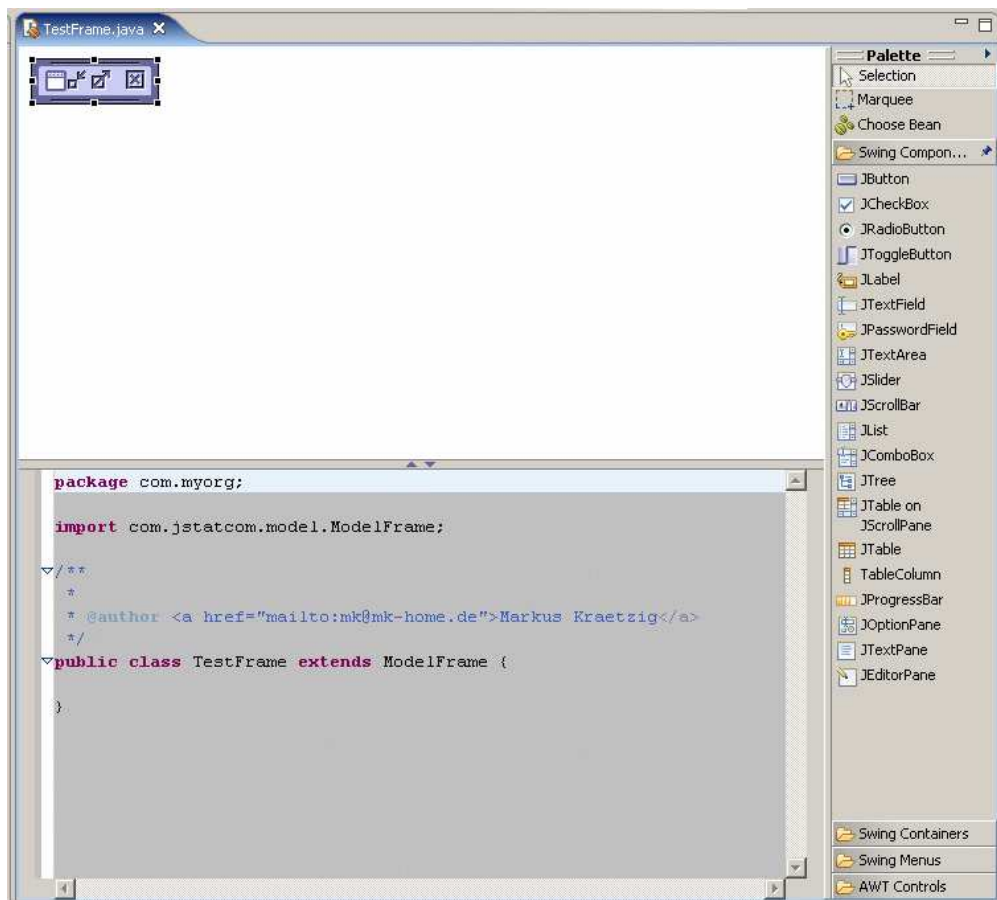


Figure 2.9: Initial display, frame needs to be resized

Having introduced the basic features of the VE, editing can be started by resizing the frame with the mouse and choosing a title. The title can be set by clicking in the header of the frame, or by directly editing the properties of that component. For that purpose, there exists the property editor of the VE, see Figure 2.10 at the bottom. It should be noted that most properties do not need to be changed from their default value, but the title property has been set to `TestFrame`, lacking more innovative ideas here. By changing the size of the frame with the mouse, the size property is automatically changed. Properties are used to customize components and help to speed up programming, especially for GUI layout.

Now that the `TestFrame` has a title and a proper size, it is required to add a panel to hold all other components. Therefore one should select the class `JPanel`

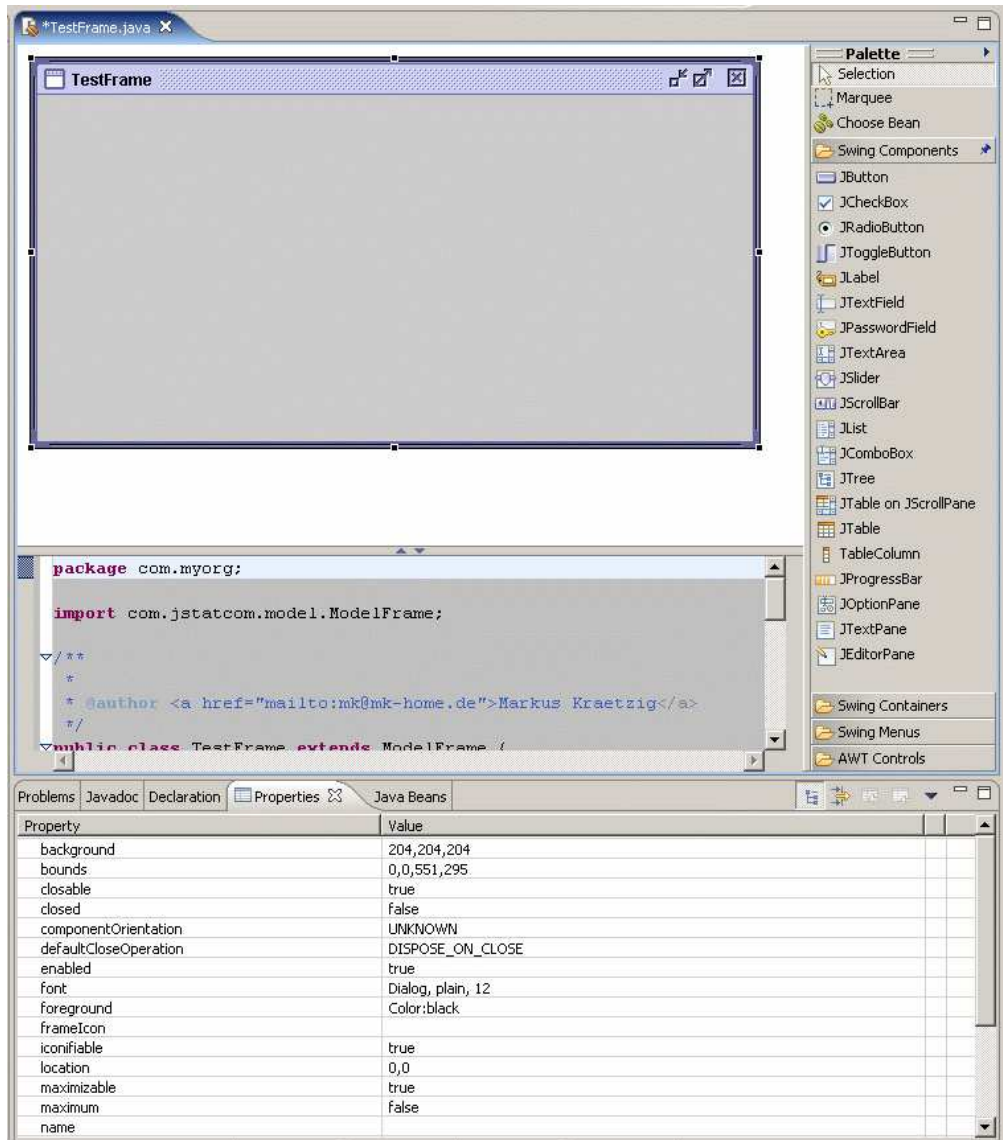


Figure 2.10: TestFrame after resize with title, properties at the bottom

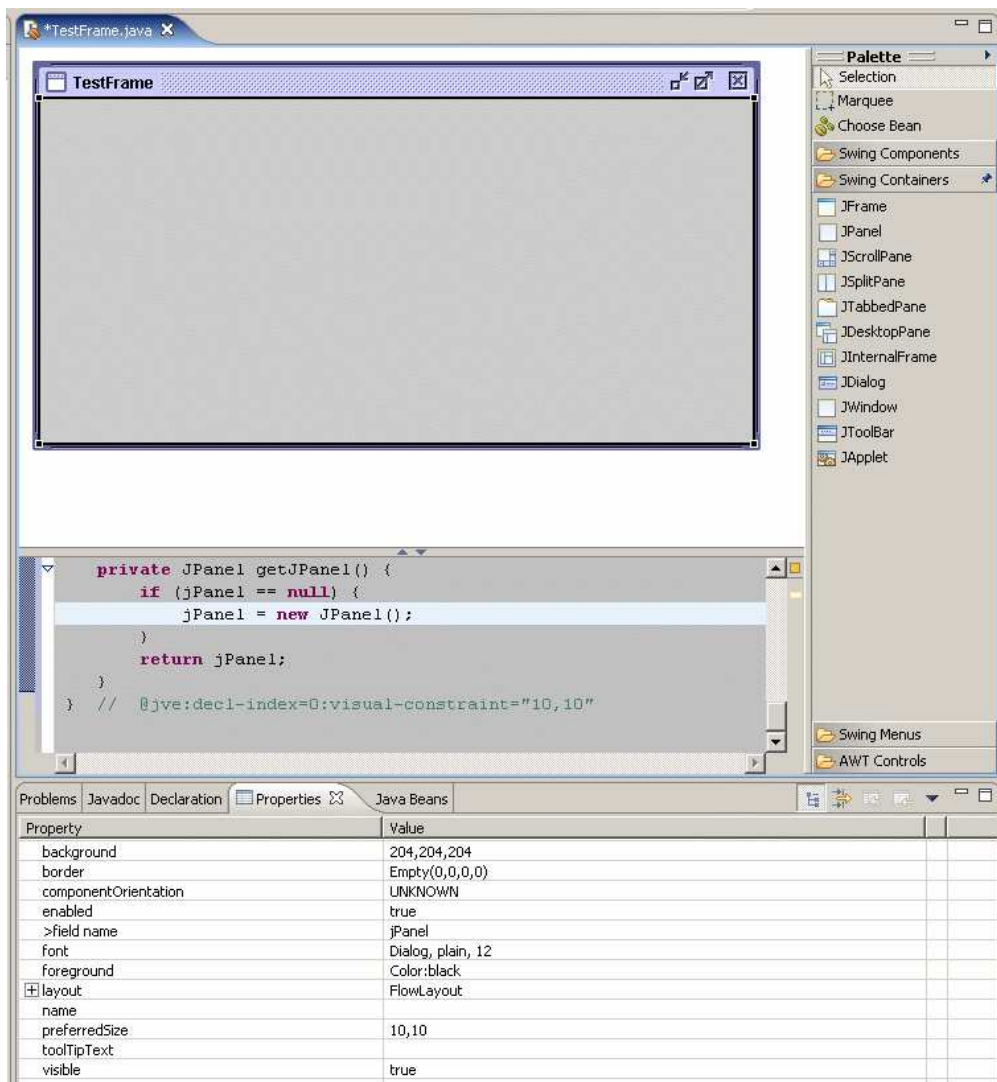


Figure 2.11: TestFrame after a JPanel has been added

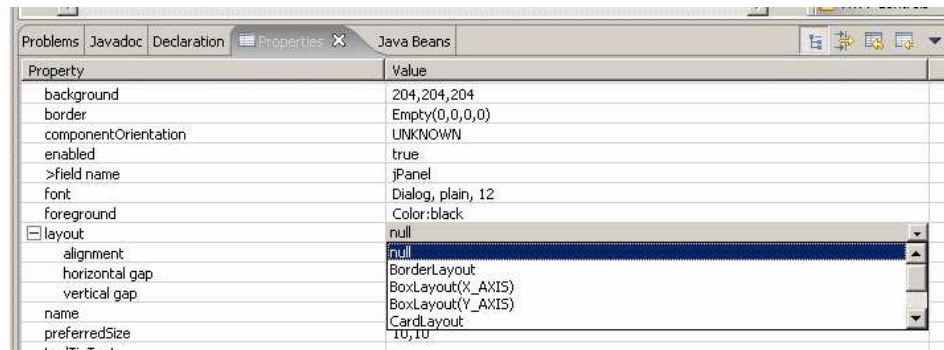


Figure 2.12: Setting layout manager to null, easier to handle for initial design

from the palette and place it in the frame. Because panels are container elements, they can be found under the *Swing Containers* tab, see Figure 2.11. Adding a `JPanel` is not a very exciting operation, because visually nothing changes. It is just an empty container. However, by looking at the Java editor one can see that some code has been generated and the property editor is now showing the properties of the added `JPanel`.

Now a small technicality is required to avoid confusions when adding new components to the panel. By default every panel has a layout manager which computes how components are being placed inside the container. It also adjusts their size and placement when the container is resized. The default manager for the `JPanel` is a `FlowLayout`. However, for this example layout management should not be discussed and it is easier to switch it off for the initial design. Therefore the property editor should be used to set the `layout` property of this panel to `null` as shown in Figure 2.12.

Having prepared the panel to hold the components, one should now select the class `TSSel` from the palette via the *Choose Bean* button. Figure 2.13 shows the selection dialog. After clicking *OK* the component can be placed on the panel inside the frame. If the enclosing frame is still not big enough, it should be resized again. After finishing this operation the new component is part of the `TestFrame`, see Figure 2.14.

The chosen component is part of the `JStatCom` framework and is used to select variables for time series models. It provides also a range of useful functions via

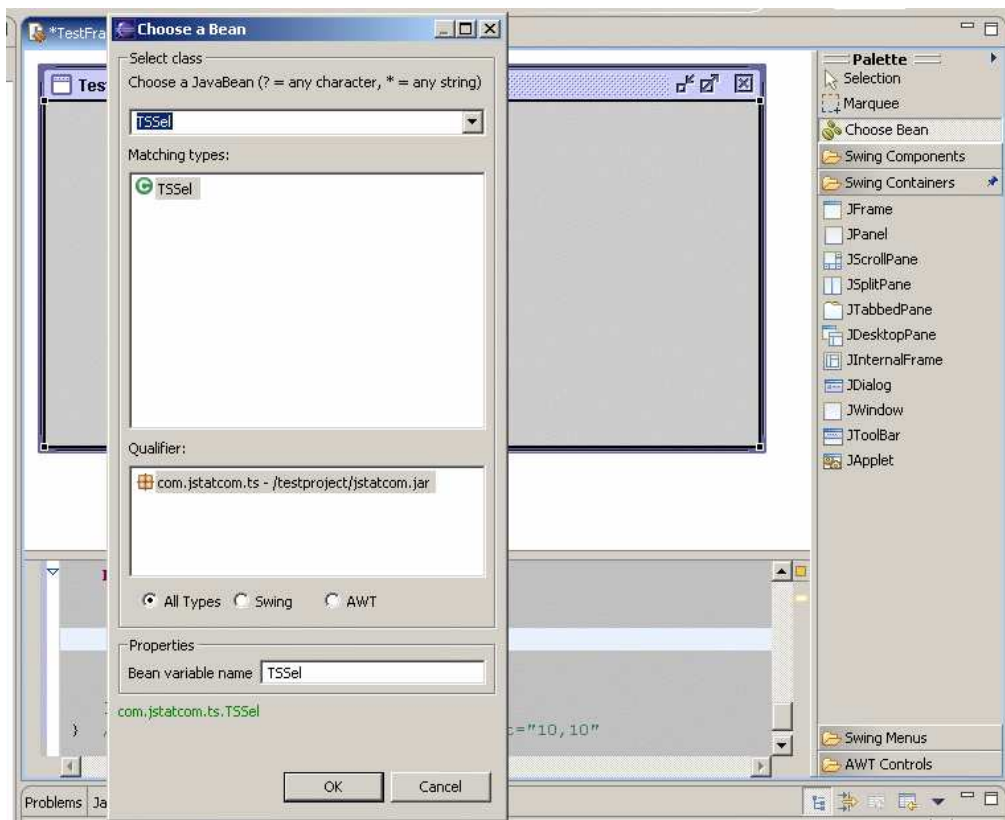


Figure 2.13: Selecting the TSSel component

a right click popup menu over the selected variables. Furthermore, the sample range can be adjusted. This way, complex functionality is integrated in the testing frame which demonstrates the advantages of component based development.

However, there are some properties that need to be set to the TSSel component with the help of the component editor. Because the class is used to select variables, it is necessary to define under which name the data objects are stored in the so called symbol table. JStatCom provides a mechanism to share data between components, a feature that is, for example, also available for Matlab GUI building in a somewhat similar way with the Matlab `guidata` function. This is only mentioned to show that similar problems arise also in other systems for GUI programming. For JStatCom, data objects must be identified via names in the symbol table. Therefore the property editor should be used to set these names, see Figure

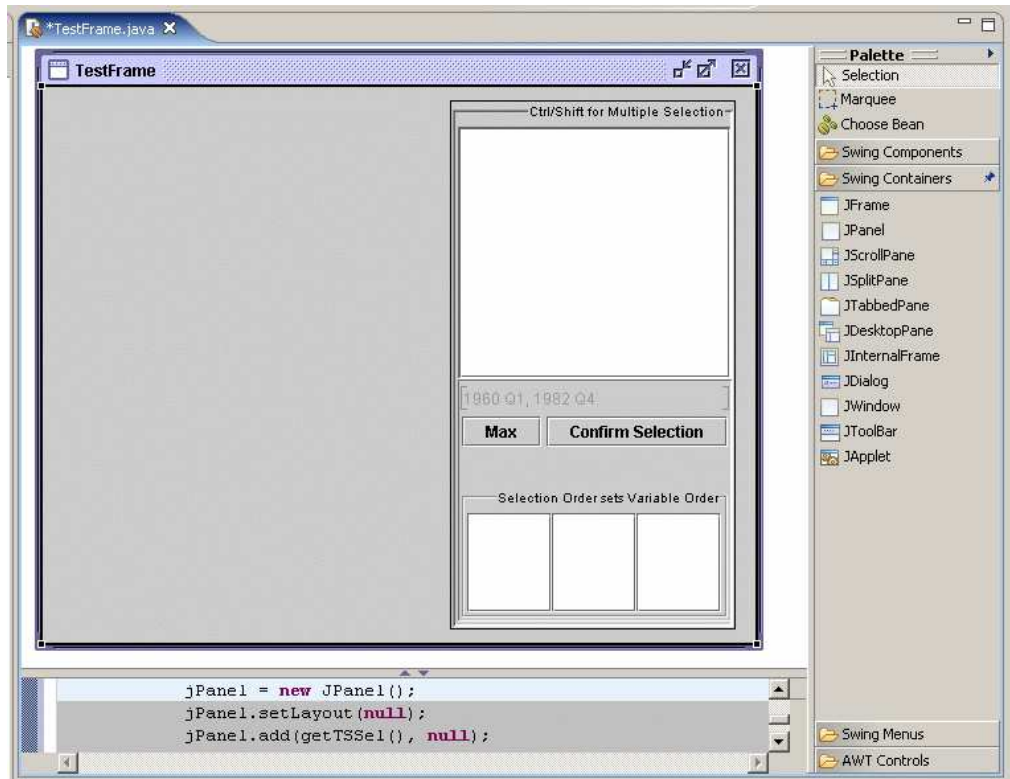
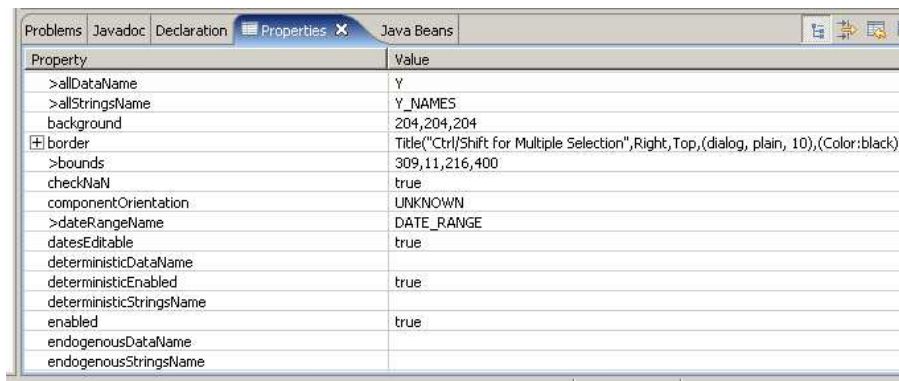


Figure 2.14: TSSel component has been placed on the panel

2.15. When the TSSel component is selected in the VE one can start editing the properties `allDataName`, `allStringsName`, and `dateRangeName`. The chosen names for this example are `Y`, `Y_NAMES`, and `DATE_RANGE` respectively.

After finishing the setup of the selection component, one still needs to add the remaining components. JStatCom provides the special text field `NumSelector` to select numbers. It should be selected and placed to the panel in the same manner as the TSSel component, see Figure 2.16.

One of the features of the number selection component is that it is easily possible to set a validating range with the help of the property editor. From Figure 2.17 it can be seen how the property `rangeExpr` is set to the string `[1, 20]`. Whenever the user specifies a lag length outside this range, an error message would appear. Furthermore, the precision of the display is set to 0 by default, which means that only integer values are shown with no decimal digits.



The screenshot shows the 'Properties' window for a Java Beans component. The window has tabs for 'Problems', 'Javadoc', 'Declaration', 'Properties', and 'Java Beans'. The 'Properties' tab is active, displaying a table of properties and their values.

Property	Value
>allDataName	Y
>allStringsName	Y_NAMES
background	204,204,204
border	Title("Ctrl/Shift for Multiple Selection",Right,Top,(dialog, plain, 10),(Color:black))
>bounds	309,11,216,400
checkNaN	true
componentOrientation	UNKNOWN
>dateRangeName	DATE_RANGE
datesEditable	true
deterministicDataName	
deterministicEnabled	true
deterministicStringsName	
enabled	true
endogenousDataName	
endogenousStringsName	

Figure 2.15: Editing properties of TSSel component

By now it should be clear how components can be added to the GUI. The remaining parts are a `JLabel` with the string `Lags`, a `JButton` with `Execute` on it, and a `ResultField` to display results. The latter component is provided by `JStatCom`, whereas the first two are standard Swing classes and can be found directly on the palette under `Swing Components`. All components should be resized and placed according to the preferences of the developer. The text of the `JLabel` and the `Execute` button can be set by clicking on the component with the mouse or by using the property editor and changing the `text` property.

One should not forget to save the file `TestFrame.java` after adding components. This can easily be done by pressing the `Ctrl-S` key combination. Saving a Java source file in the IDE automatically invokes the compilation.

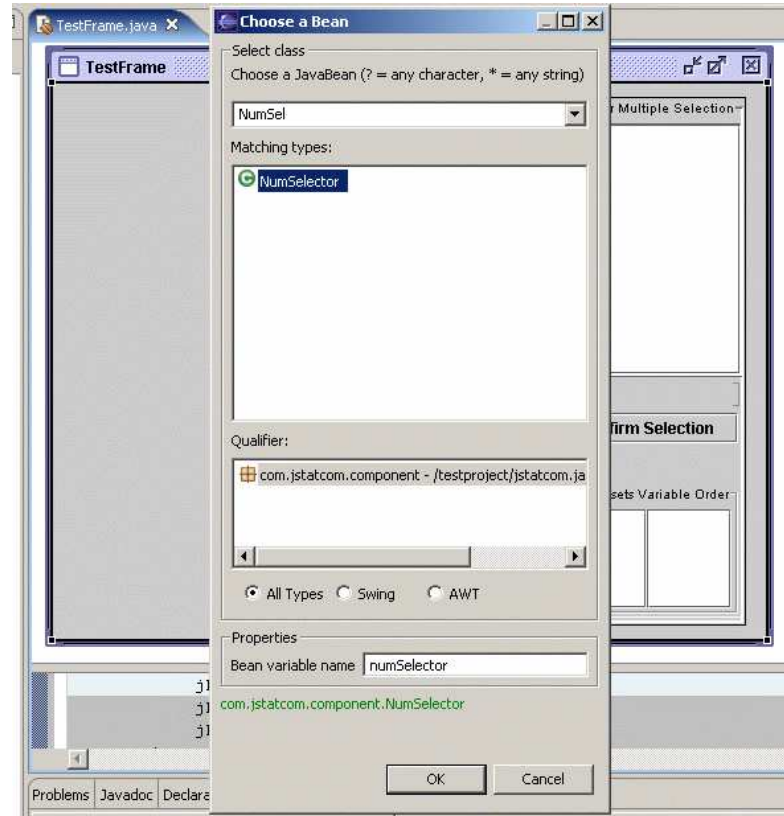


Figure 2.16: Selecting the NumSelector component for number input

Property	Value
foreground	Color:black
horizontalAlignment	LEADING
intType	false
location	58,52
name	
number	0
numberFormatType	(numberFormatTypes\$1)
numberRange	(numberRange)
precision	0
preferredSize	11,20
>rangeExpr	[1,20]
size	55,20
symbolName	

Figure 2.17: Setting a validating range [1, 20] to the NumSelector component

2.2.8 Step 7: Add an Action to the Execute Button

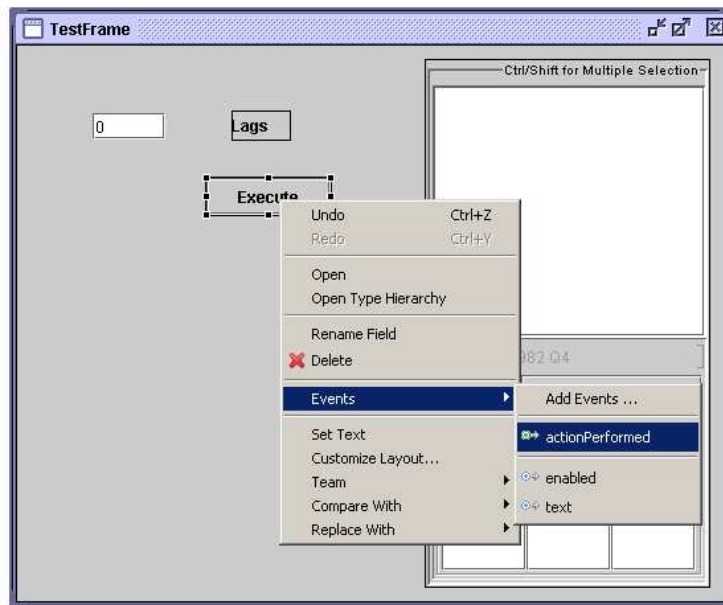


Figure 2.18: Placing a JButton and creating an action

After the basic layout of the GUI is finished, it is necessary to program an action that is invoked when the *Execute* button is pressed. Without this, nothing would happen at all. The VE also supports this task via a menu that appears on a right click over the button, see Figure 2.18. The item *Events - actionPerformed* should be chosen to install a default action to that component.

Figure 2.19 shows the Java code that has been generated by this operation. There is an `actionPerformed` method which just prints out some default string. This method must later be programmed 'by hand' to gather the input from the GUI controls, to call the external Gauss procedure, and to print the results.

2.2.9 Step 8: Add the Module to the Main Application Frame

Before it is described how the algorithm for the ARCH-LM test is invoked one would probably like to check what has been accomplished so far. Therefore the module is integrated in the JStatCom application framework and can be run. For

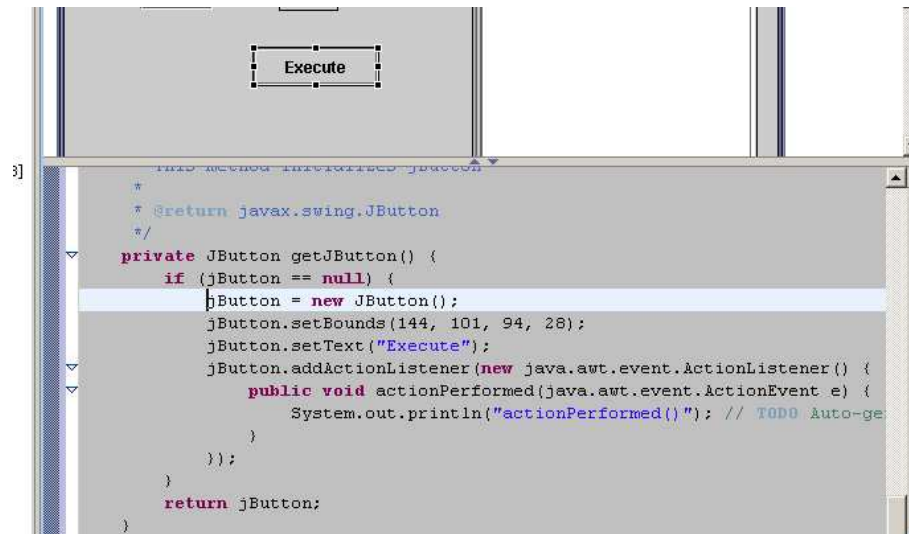


Figure 2.19: Default action handler for execute button

this, no further Java programming is required, only two configuration files need to be adjusted.

Figure 2.20 shows the file `modules.xml` in the text editor of the IDE. It should be opened via the right mouse menu. The format of that file is XML and it already contains some example entries which have been commented out. To integrate the newly created module without extra Java programming, the line

```
<Module class="com.myorg.TestFrame"/>
```

should be added and the file should be saved. Note that the slash at the end is needed. Afterwards the file `app.bat` has to be edited to add the location of the classes to the classpath, see Figure 2.21. The folder with the compiled classes is `bin` by default, therefore the line

```
SET CP = bin;%CP%
```

must be added. Finally, one can now start the application by clicking on the file `app.bat` from outside the IDE software. It will invoke the application and the module can be accessed via the menu item *Modules*. Figure 2.22 shows the running program. The module is shown together with the default data import frame, which is a standard feature of JStatCom. Although nothing happens yet

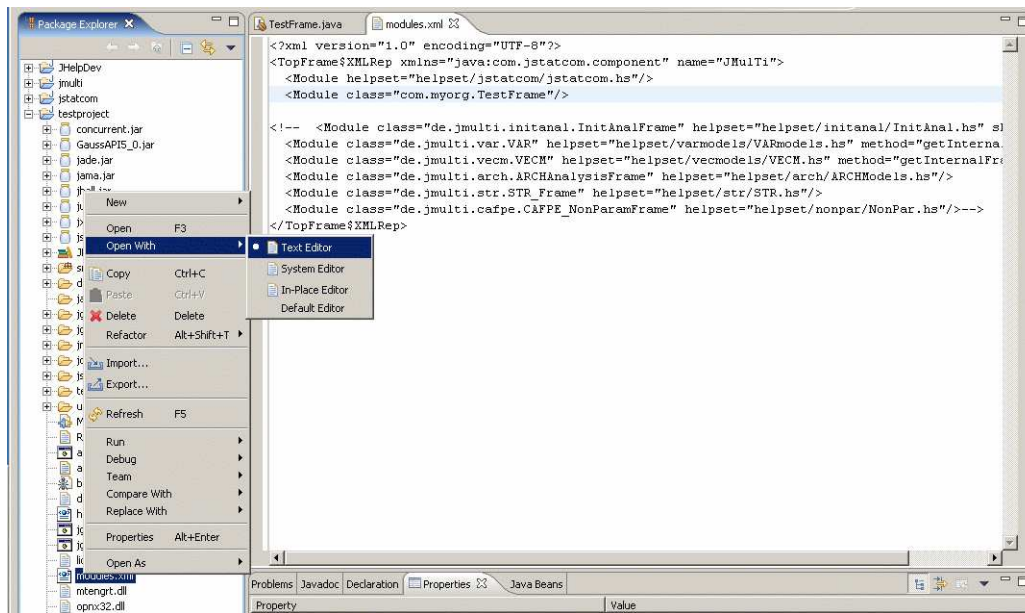


Figure 2.20: Editing `modules.xml` to insert `TestFrame` to list of modules

when *Execute* is clicked, except that some string is printed to the standard output, there are already a lot of features available provided by the framework.

For example, it is possible to import datasets, to use the time series calculator for combining variables with arithmetic operations, and to edit and transform series that have been read in. It is also possible to create dummy variables conveniently. In the module `TestFrame` one can select variables, and the text field for the number of lags validates input according to the interval set. All this general functionality is provided by `JStatCom` and it is now up to the developer to use this infrastructure for implementing calls to specific algorithms.

2.2.10 Step 9: Integrate Gauss Algorithm

After having programmed the GUI in Java, one has to prepare the Gauss code with the algorithm. `JStatCom` provides a communications interface to an installed Gauss, but the code has to be compiled as a `gcg` file first and it has to be put in a special location. The Gauss procedure to compute the ARCH-LM test (Engle (1982)) is given in the code example. Typically, this procedure already exists and

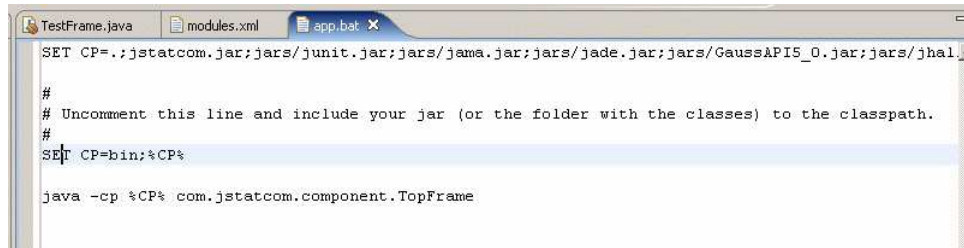


Figure 2.21: Adjusting the classpath in the app.bat script

is provided by a researcher who has domain-specific knowledge. The Java programmer does not necessarily need to know the inner workings of the algorithm, but she must of course understand its interface and how to design the GUI for it. The interface of a GAUSS procedure includes the name, the arguments, and the return values. It must be clear how the input parameters have to be assembled and what the procedure gives back.

```

/**
 * ARCH-LM test (Engle 82).
 *
 * resid - Tx1 vector with residuals
 * q - number of lags to include in test regression
 *
 * result - Chi2_stat~prob_chi2~F_stat~prob_F
 */
proc(1) = archlm_mytest(resid, q);
  local y,ylags,Tnew,b,sigma,rsq,f,XX,r,prob_chi,prob_F;
  y      = (resid - meanc(resid))^2;
  ylags  = shiftr((ones(1,q) .* y)',seqa(1,1,q),-exp(20))';
  ylags  = delif(ylags,ylags[.,cols(ylags)] .== -exp(20));
  Tnew   = rows(ylags);
  y      = y[rows(y)-Tnew+1:rows(y)];
  ylags  = ones(rows(ylags),1)~ylags;
  b      = inv(ylags'ylags)*ylags'*y;
  sigma  = (y-ylags*b)'(y-ylags*b)/rows(y);
  rsq    = 1-sigma/((y - meanc(y))'(y - meanc(y))/rows(y));
  XX     = ylags'ylags;

```

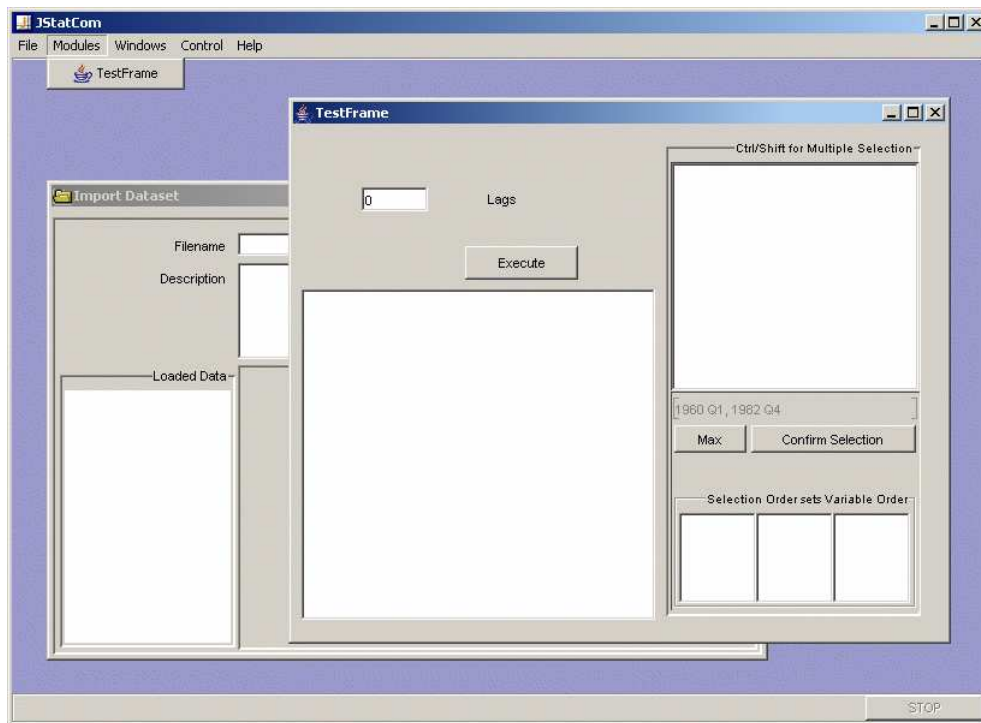


Figure 2.22: Running application with new TestFrame module, execute method still needs to be coded

```

R      = zeros(q,1) ~ eye(q);
F      = (R*b)'inv(R*inv(XX)*R')*(R*b) / (q*sigma);
prob_F = cdfFc(F,q,rows(ylags)-rows(b));
prob_chi= cdfchic(rows(ylags)*rsq,q);
retp(rows(ylags)*rsq~prob_chi~F~prob_F);
endp;

```

It is assumed that the code is stored in the file `mytest.src`. This file should be put in the subdirectory `jgauss/src` of the project directory. There are also some other source files which are needed by the communications library. One should be aware that Gauss procedure names should be unique across all files, therefore the name `archlm_mytest` was chosen. The convention is that the filename is used as a suffix to the descriptive name of the procedure.

As a last step, one must compile the source code together with the other files in the directory to a file `jgauss.gcg`. This could be done manually with Gauss, but

JStatCom provides the batch script `jgauss_gcg.bat` to make it easier to compile also a large number of source files. For this the file `jgauss/compile.xml` has to be edited:

```
<?xml version="1.0" encoding="UTF-8"?>
<GCGSet xmlns="java:com.jstatcom.engine.gauss" gcgfile="jgauss.gcg">
  <GCGSet$SRCFile filename="src/jmplot.dec"/>
  <GCGSet$SRCFile filename="src/jgauss.src"/>
  <GCGSet$SRCFile filename="src/jgrte.src"/>
  <GCGSet$SRCFile filename="src/tools.src"/>
  <GCGSet$SRCFile filename="src/plot.src"/>
  <GCGSet$SRCFile filename="src/mytest.src"/>
</GCGSet>
```

Only the last line has been added. By clicking on `jgauss_gcg.bat` all source files are compiled and the file `jgauss/jgauss.gcg` is being created. When the batch script is started first it is likely that some settings for the Gauss software have to be set. Those settings could also be set manually in the file `jgauss/engine_config.xml`. JStatCom needs to know the correct version number and the location of the Gauss executable.

2.2.11 Step 10: Implement the Execute Routine

Now that the Gauss procedure `archlm_mytest` is recognized by the Gauss engine, it is possible to implement the Java call to that procedure. It should be mentioned that JStatCom has its own data model to represent numbers, matrices, strings, string arrays, dates, and date ranges. All data classes start with the prefix `JSC`, which is short for JStatCom. For example, `JSCNArray` represents an $m \times n$ number array, whereas `JSCInt` stores an integer scalar. The basic steps for the execute call are:

1. Get Gauss engine instance

```
Engine engine = EngineTypes.GAUSS.getEngine();
```

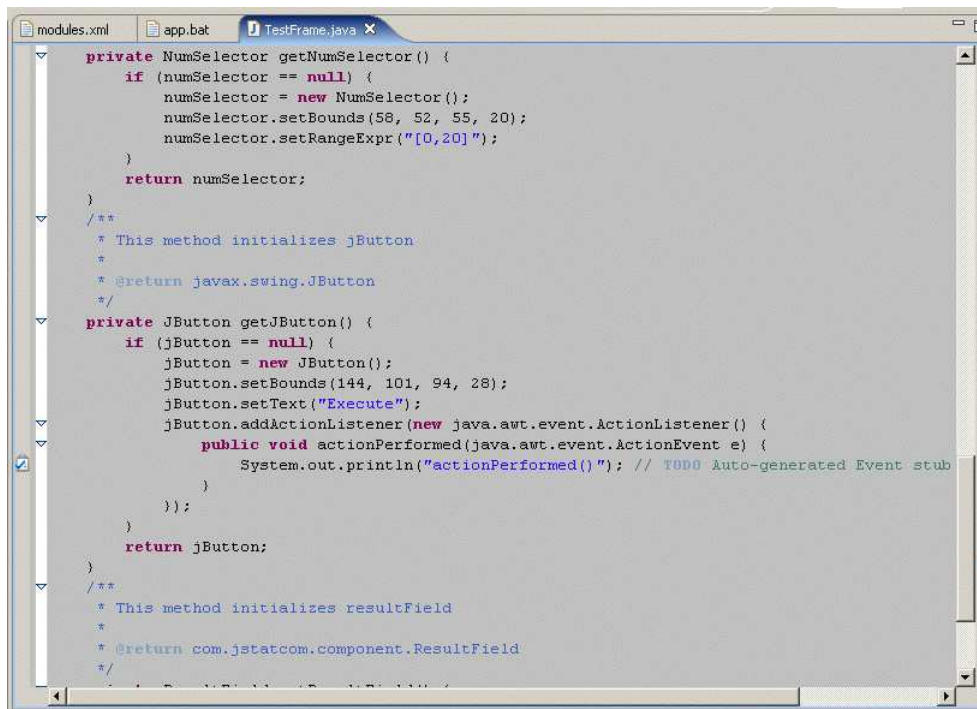



Figure 2.23: Open TestFrame in Java editor, more convenient for manual coding than visual editor

2. Retrieve the input data objects (// starts a Java comment)

```

// retrieves shared symbol by name
JSCNArray y = global().getSymbol("Y").getJSCNArray();
// creates new integer object with the selected lags
JSCInt lags = new JSCInt("LAGS", getNumSelector().getIntNumber());

```

3. Create an empty data object for the result of the computation

```

JSCNArray result = new JSCNArray("TESTRESULT");

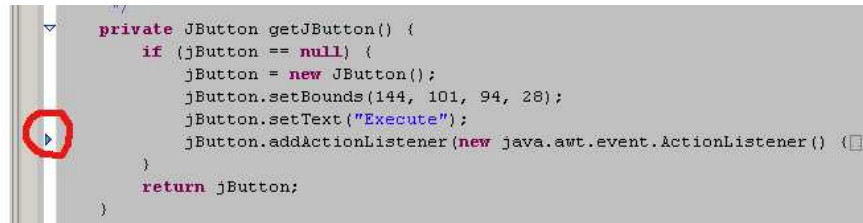
```

4. Call the Gauss procedure

```

// call the procedure with input and output parameters
engine.call("archlm_mytest", new JSCData[]{y, lags},
            new JSCData[]{result});

```

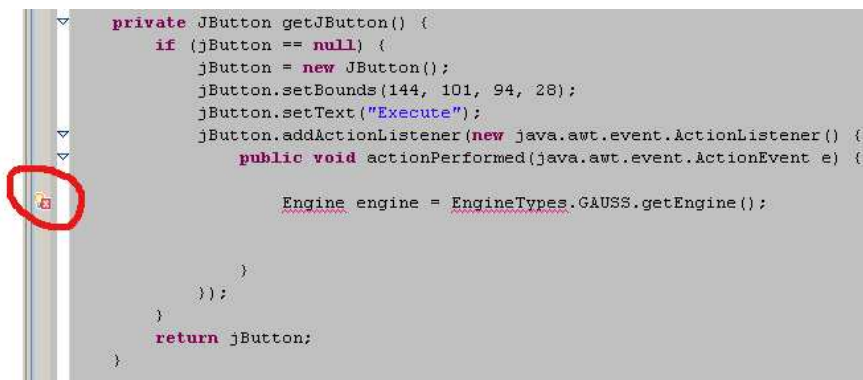


```

private JButton getJButton() {
    if (jButton == null) {
        jButton = new JButton();
        jButton.setBounds(144, 101, 94, 28);
        jButton.setText("Execute");
        jButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                Engine engine = EngineTypes.GAUSS.getEngine();
            }
        });
    }
    return jButton;
}

```

Figure 2.24: Hidden method body that should be expanded by clicking on arrow to the left



```

private JButton getJButton() {
    if (jButton == null) {
        jButton = new JButton();
        jButton.setBounds(144, 101, 94, 28);
        jButton.setText("Execute");
        jButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                Engine engine = EngineTypes.GAUSS.getEngine();
            }
        });
    }
    return jButton;
}

```

Figure 2.25: Red underline and symbol to the left indicating some compilation problem

5. Display the result and set it to the symbol table

```

getResultField().append(result.display());
global().set(result);

```

Some manual Java programming is required here. This is done more easily with the Java editor instead of the visual editor in the IDE. Therefore the VE should be closed and the Java editor should be opened via the right click menu over the `TestFrame.java` file, see Figure 2.23. It should be noted that by default this method is hidden in the editor and should be expanded first, see Figure 2.24. When editing is started, one could use the automatic expansion mechanism by pressing the `Ctrl-Space` keys together. If any typos occur, the editor indicates this by underlining the respective element and showing an error symbol at the left edge, see Figure 2.25. Often problems occur because a class name is only

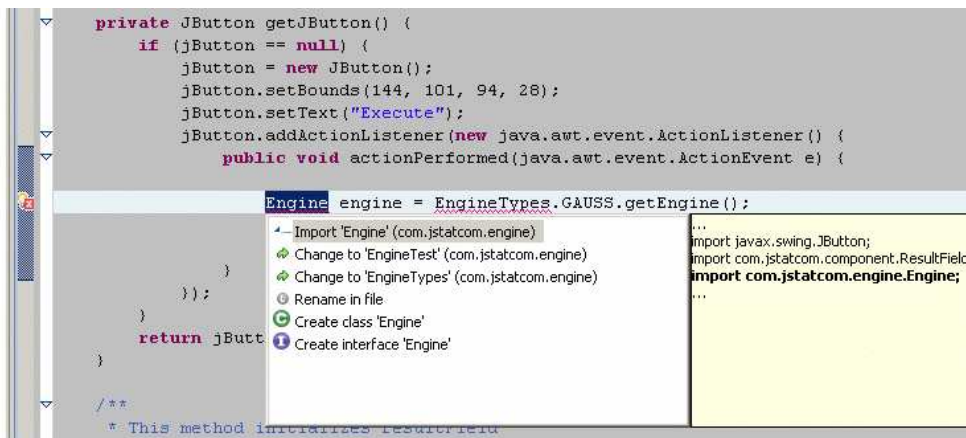
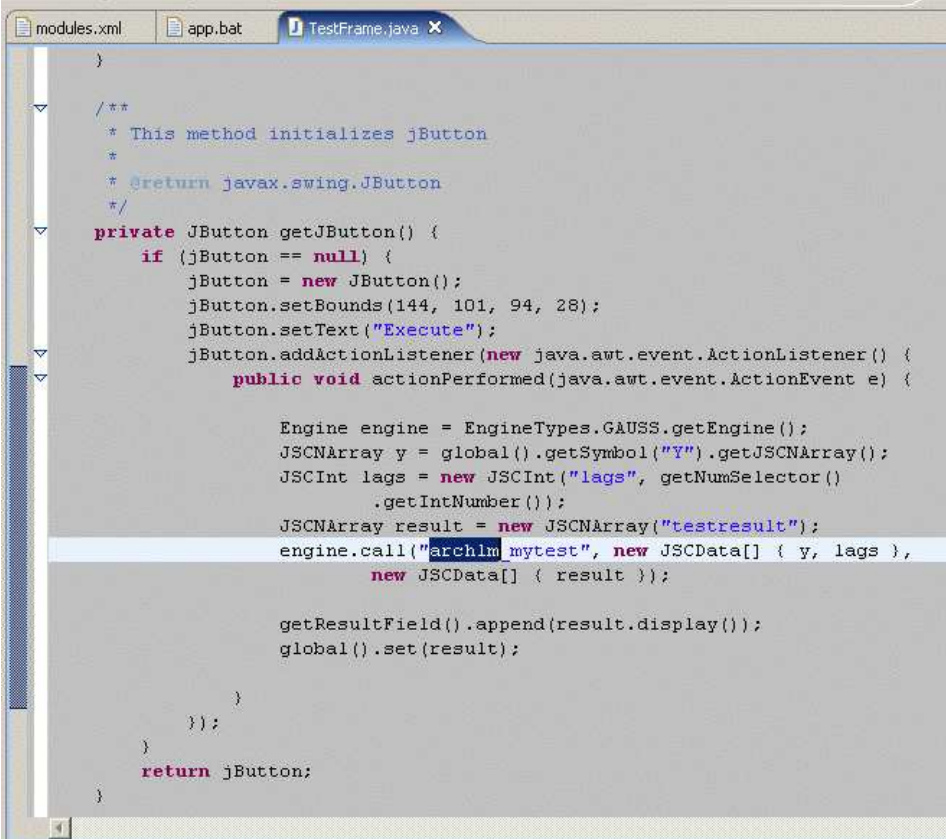


Figure 2.26: A left mouse click on the error symbol gives a menu with possible error fixes, the first option should be chosen here

recognized after it has been imported. However, by clicking on the error symbol one can easily fix those problems automatically, see Figure 2.26. The full method body is seen in Figure 2.27.



```
modules.xml  app.bat  TestFrame.java x
}
/**
 * This method initializes jButton
 *
 * @return javax.swing.JButton
 */
private JButton getJButton() {
    if (jButton == null) {
        jButton = new JButton();
        jButton.setBounds(144, 101, 94, 28);
        jButton.setText("Execute");
        jButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {

                Engine engine = EngineTypes.GAUSS.getEngine();
                JSCNArray y = global().getSymbol("Y").getJSCNArray();
                JSCInt lags = new JSCInt("lags", getNumSelector()
                    .getIntNumber());
                JSCNArray result = new JSCNArray("testresult");
                engine.call("archlm_mytest", new JSCData[] { y, lags },
                    new JSCData[] { result });

                getResultField().append(result.display());
                global().set(result);

            }
        });
    }
    return jButton;
}
```

Figure 2.27: Implementation of the execute call

2.2.12 Step 11: Check running Module

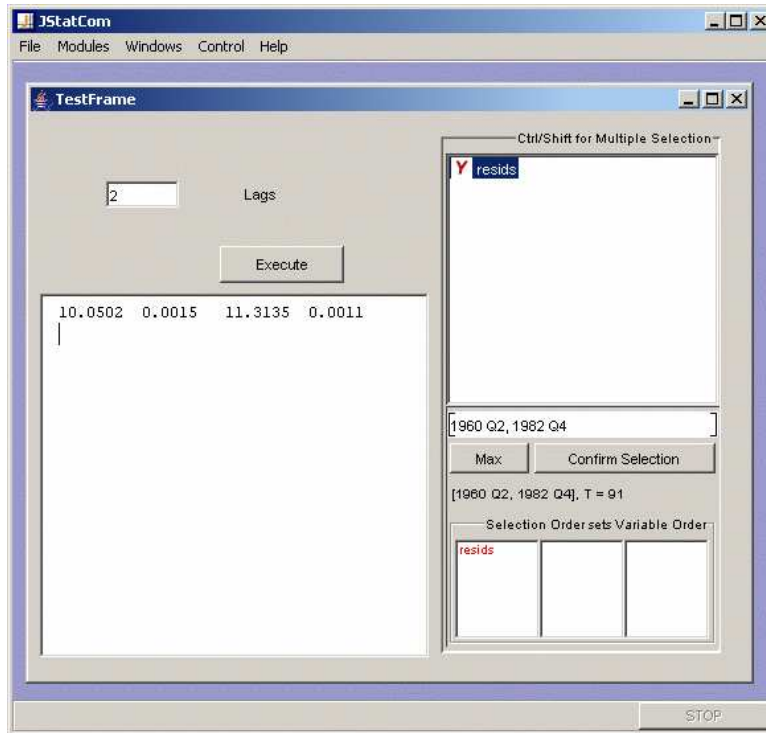


Figure 2.28: Running module with output from computation

Now that everything has been programmed, the module should be checked again via clicking on the `app.bat` script. After a time series has been imported and selected, a click on the *Execute* button will invoke the computation and the result will be printed to the text area. The resulting numbers include the Chi-squared and F statistics together with the corresponding p-values, see Figure 2.28. Furthermore, the result has been set to the symbol table of the module. This can be checked with the Symbol Control which is accessible via the *Control* menu, see Figure 2.29.

Another small customization of the new application could be done by editing the file `app.properties` in the project directory. It is possible to change the title, the splash screen, the version number, as well as the about information of the software.

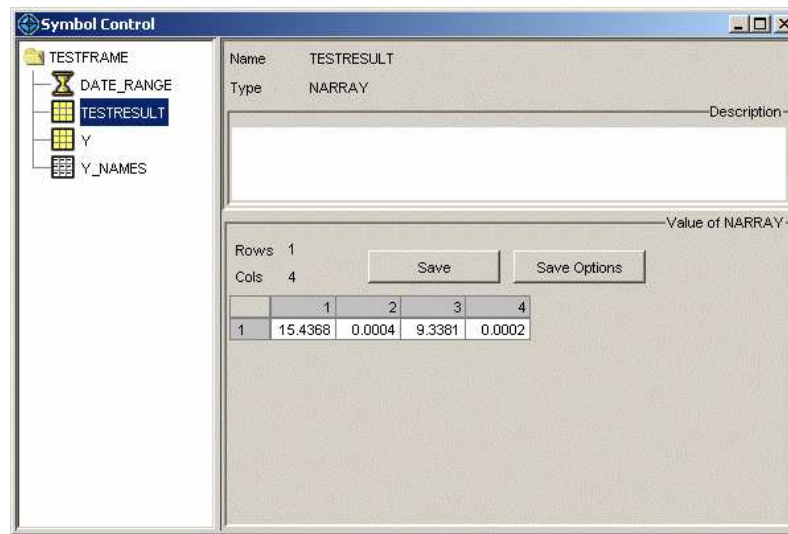


Figure 2.29: Symbol Control after the computation has finished

2.3 Final Remarks

This example should have motivated the use of JStatCom, although it has only scratched the surface of what can be done with the framework. The module that has been programmed could still be refined in several ways. There should be an input check before the procedure call is actually carried out. Output formatting could be improved as well. Furthermore, one should use a layout manager to make the GUI look more professional, especially when it is being resized. Another important point is that the module consists of only one class, which is appropriate for this simple example, but which should be changed if more features are to be added. In that case one should consider creating a separate class which only holds the panel. It could then be added to the frame. If more features are implemented, the frame could get a tabbed pane or a menu bar, and it could hold many panels instead of just one.

An important point is that the execute routine in its current form is defined inside the GUI class. In general this is not recommended and JStatCom provides the so called PCall system to encapsulate the calling logic in separate classes. This has the advantage that the procedure calls could be maintained separately and that they could also be used independently of the GUI classes.

Chapter 4 gives a number of advanced programming examples with JStatCom together with guidelines for object-oriented design with that system. However, having this introductory example running is a good starting point for many possible enhancements.

Chapter 3

Design and Implementation

3.1 Documenting a Software Architecture

This chapter contains the architecture documentation of JStatCom. Although there exist many ways to document software systems, reaching from very formal approaches, like Acme (Garlan et al. (2000)), to purely textual descriptions, it seems that common standards are emerging and are being adopted, for example IEEE (2000). However, given the range of different possible notations, these documentation standards cannot be seen as strict rules, but rather as a rationale that should help to record all relevant aspects of a system in a form that is accessible to project stakeholders.

A general problem of non trivial software systems is, that they cannot be understood by using just a single view or abstraction, but only by applying a number of different perspectives. Sometimes it is important to understand how different components of the system collaborate, sometimes the focus is on the class structure of a module. It is also of interest who is addressed by the documentation, because it makes a difference whether a general overview is given to a customer or whether a new developer is introduced to the inner workings of a subsystem. The UML language provides graphical constructs to represent many different views into a complex software system (Booch et al. (1999)). It is possible to describe almost every feature of the framework graphically, but that would render the resulting diagrams very complex and thus useless as a way to communicate only

relevant aspects of the architecture. Therefore software documentation must focus on the important details and relations in each view. An up-to-date discussion of this topic is given in Clements et al. (2003), and this chapter will follow the suggestions given there. A short overview of the used UML constructs can be found in the appendix.

The documentation of JStatCom consists of several views. Each of them focuses on specific details of the system and is organized into further view packets that chunk information on different areas of the system. However, first it is necessary to provide information that binds all different views together, and to lay out a roadmap on how to use the documentation.

3.2 JStatCom System Overview

JStatCom is a software framework for data based analysis that can be used to develop applications for econometrics and statistics. This chapter describes how the software is implemented and how it can be used and extended. Developers do not necessarily need to understand every aspect of the system to build applications with it, but only the main concepts. An indispensable source of information is the API documentation, which conforms to the requirements for class specifications (Berard (1993)).

This framework is an object-oriented system that consists of classes implemented in Java, therefore applications using it should preferably be written in Java as well. It can be used with several external execution engines to which it provides a standardized communications interface. Alternatively, one could apply Java libraries for the numerical computations. JStatCom itself does not contain advanced numerical algorithms but serves as an architectural layer that can connect different libraries and toolkits. Its intended use is the creation of graphical user interfaces, but parts of it might also be used by non-graphical Java applications. The framework has a data model to represent types commonly needed in scientific computing, like numbers and arrays. This data model is supported by various components to display and manipulate its contents. One may also extend this model by defining new types. However, these extensions are not the standard usage of the framework.

I will therefore distinguish between two basic activities:

- *Usage* - the standard case; functionality of JStatCom is accessed via the API; requires only a basic understanding of the framework's inner workings and average programming skills in Java
- *Extension* - the advanced case; functionality of JStatCom is altered by overwriting certain classes, or new functionality is added by providing additional components or classes; requires a good understanding of the related mechanisms within the framework and more advanced programming skills

Extension guarantees that the framework can also adopt to situations, where no standard solution exists yet. However, some simple extensions might occur quite often, for example if a new renderer for the contents of a data table is defined. More advanced examples would be to define a new data type that is recognized by the internal data management system or a whole new engine communications scheme. In principle these extensions could be integrated in the core framework, if they are of general enough interest. Extension and Usage correspond to the terms white-box reuse and black-box reuse respectively (Pree (2000)).

3.3 How Stakeholders Can Use the Documentation

Here the roles that stakeholders might have are listed together with a description on how to use this software documentation to address the relevant concerns.

- *GUI Developer*: For GUI development, it is necessary to understand the components that are provided by the framework, as well as the data model and the engine communication. Section 3.3.1 describes the typical development steps together with the most important related view packets that should be consulted. In addition to the software architecture documentation, the API specification is the most valuable source of information. The latter contains very detailed information on how to use specific classes and interfaces, but is not very good at providing an overview of the system.

- *Scientist*: Algorithms for a problem can be developed largely independently of JStatCom. However, it is necessary to have a basic understanding of the engine communications scheme that will be applied. Reading the respective view packet for the applied execution engine should be sufficient. It is important to know, which data types are supported, how output can be handled and how the algorithms can be invoked from the framework.
- *Framework Developer*: To maintain and extend parts of JStatCom, a thorough understanding of all involved modules is required. Therefore all view packets that contain the module under consideration should be read.

3.3.1 Typical Development Steps

Creating applications with JStatCom consists of a number of steps that are often similar. Table 3.1 gives an overview of the basic methodology and links to the view packets (VP) that provide further details for each step. Not all mentioned view packets are always relevant, but only if the respective functionality is actually needed. View packets that are not mentioned have been left out, because they are less likely to be directly used by the GUI developer. However, for advanced usage scenarios they might well be included too. This overview should only serve as a starting point.

It is assumed that an IDE is used that supports Java development. Using such a tool is not required, but it increases productivity dramatically, especially when projects grow and get more complex. There are commercial and open-source tools available from which one can choose freely. Excellent, cost efficient tool support is one of the advantages of using the Java language.

Task	Documentation
1 download and unpack JStatCom	JStatCom (VP 1)

-
- 2 set up a Java project with your favourite IDE, put `jstatcom.jar` and all jar archives from the `jars` subdirectory in the classpath that is used by the development tool IDE specific documentation and homepage of JStatCom
-
- 3 create the top level component by subclassing `ModelFrame`, choose an appropriate name and title, and compile it (typically done automatically by IDE) Symbol Management (VP 5)
-
- 4 put the fully qualified classname of the new module frame in the file `modules.xml` and start the application with the batch file `app.bat/app.sh` Application (VP 21)
-
- 5 implement all panels, use subclasses of `ModelPanel` if access to shared data is needed Symbol Management (VP 5), Symbol Event System (VP 6), Selection (VP 18), Components (VP 20), Data Table (VP 22), Equation (VP 23)
-
- 6 implement calls to the possibly external modules implementing the math algorithms, put all procedure modules that are used in the engine-specific subdirectory, set up and run sets of automated unit tests for all engine calls Data Model (VP 2), Type System (VP 3), Engine (VP 8), specific engines (VP 9-13), PCall (VP 14), Time Series (VP 15)

-
- 7 create a deployable version, possibly with a build script, the simplest way would be to zip the project directory
-

Table 3.1: Typical development steps

One should also provide documentation for the user as a helpset. The add-on tool JHelpDev, which is described in Appendix B, can be used to create a JavaHelp set from a directory with HTML files. It can then easily be integrated in the application, even with context-sensitive help. How this is done is described in Chapter 4. Together with Latex2html or other converters, one could write the help system completely in Latex, which is especially useful if many formulas are used.

3.4 Background, Rationale, and Design Constraints

This section records the requirements aspects of JStatCom that have led to the major architectural decisions. Most of the feature requests resulted from the experiences when developing the software JMulTi.

3.4.1 Operational Context

JStatCom is expected to be used in a wide range of data based analysis applications. Therefore it must be flexible enough to be extended or customized to adopt to different environments. It must not make strong assumptions about the structure of the scientific models to be used, about domain-specific data types, the format of import files, or how numerical algorithms are implemented. The framework must be easy to use, because it should be applied by scientists, instead of professional software developers. A strong emphasis is put on GUI creation. JStatCom should provide a variety of components that can be plugged together with the support of visual programming tools. The data model must interact with the GUI components in a standardized fashion. Data processing must support collection and

¹The URL for the ANT build tool is *ant.apache.org*.

specification of input data, preliminary transformations and validity checking, as well as convenient output representation and formatting.

3.4.2 Key Data Management Features

In order to support a wide range of different data types, JStatCom should rely on a metadata model. Core attributes are standardized for all data types. Each data type may define additional attributes. New types can be dynamically added to the system.

To facilitate GUI design, components within a certain scope must have access to a common shared data repository. This would decouple the modules belonging to one scientific model by reducing the number of direct dependencies among them. On the other hand, access to the shared data must be confined to the components within a distinct modelling context. For example, panels for estimation, residual analysis and structural analysis in a VAR modelling context share common data from the model specification via a shared data repository. Components from other models, say from an STR model, do not have access to that repository, instead they use their own data pool. This requires the concept of defining scopes for different data repositories.

Furthermore, the data model must interact with the GUI components via events to notify interested listeners about changes in the underlying value. These changes might trigger various actions, for example updating a display or enabling/disabling some element of the user interface.

Another desirable feature would be to represent the state of all shared data repositories in a graphical component. This could be used to inspect intermediate results or to export some variable to a file. It can also help with debugging. Such a data control system must work automatically without any additional programming effort.

3.4.3 Key User Interface Features

The user interface components must support key tasks involved in specifying, estimating, and evaluating statistical models. Special requirements are input validation against range bounds, adequate rendering and editing of the used types, as

well as efficient representation of potentially large data arrays. All components must be highly configurable and must support the use of visual programming tools. For this reason a component model must be adopted, which is JavaBeans for Java components. There must be sufficient default functionality provided to set up a new application quickly. However, it must also be possible to change existing default behaviour according to specific needs.

Another requirement is the availability of domain-specific user components for a certain class of models. JStatCom provides a distinct variable selection mechanism for time series models. However, usage of these components is not required, instead custom components can be applied.

A general mechanism for project management, as well as data import from files is required and must be supported by the user interface. This mechanism must be flexible, because applications might require different file formats and will store a different set of project settings.

Because algorithms for data based analysis often can fail if the input has special features, errors resulting from a computation are not unlikely. Therefore a powerful automated error handling and logging scheme must be provided to give detailed feedback about potential causes of failure.

3.4.4 Key Interoperability Features

Because scientific algorithms may be programmed in various ways, JStatCom must operate with many different software products to enable scientists to integrate a rich set of features without the need to rewrite complex algorithms. In order to keep programming with JStatCom simple, a generalized interface to call procedures from different sources must be implemented. This interface should hide purely technical aspects of the respective calling mechanism from users of the framework.

Furthermore, input and results of procedure calls must conform to the internal data model, without the need to transform data to engine-specific types.

Applications based on JStatCom should be portable to a number of different operating systems. This should be achieved by the choice of the programming language, but it must as well be supported by the engine communications schemes.

Especially in scientific environments, the Linux and Unix family of operating systems is quite popular, therefore those systems should be supported in addition to Windows.

3.4.5 Key Design Features

Conceptual Simplicity

The framework should be adopted by scientists, rather than software engineers. Therefore conceptual simplicity is required when programming with it. This can be achieved by providing not only the needed functionality, but also a reusable design, which can be adopted by any application in the problem domain. This is the essential benefit of using a framework. It relieves developers from the complex task of setting up a new class model for every application. Instead, distinct design guidelines can be followed, which are standardized ways to proceed when setting up a new application. If the guidelines are used, then all applications based on the framework will have a similar structure, thus reducing the conceptual burden to understand and maintain them.

JStatCom design guidelines should help developers especially with the following tasks:

- defining, managing and documenting scientific models with potentially many variables
- creating well structured, maintainable GUI components
- using shared data repositories with appropriate scopes
- programming calls to algorithms that can be reused internally and that are easily testable with JUnit
- defining test cases and managing a growing set of automated unit tests
- managing modules for different models
- creating and integrating help documents for scientific models

Testability

Programming errors are not the exception, they occur all the time. Modern programming environments make it extremely easy to find and correct syntactical and semantic errors. However, logical errors are often hard to detect. In current software engineering practice, a widely used strategy to guard against programming errors is to use unit tests, see Beck (1999) for a motivating and very informal introduction. Except in trivial cases, tests cannot prove the absence of errors, as Dijkstra (1969) pointed out. They nevertheless help to discover errors very quickly, especially if they are run automatically after every change in the software. This so-called regression testing is extremely helpful to find errors that have been introduced by a refactoring or after new features have been added. The strategy leads to a path, where code can be changed and extended without breaking existing features. A feature is defined here as functionality that is tested according to a given specification which must be agreed on before.

Most tests are based on simple assertions, meaning that the actual state of a variable is compared to its assumed state. For an algorithm, this could mean that the result of a computation for a given test dataset is compared to the already known correct results. The most important aspect is that the test is automatically run together with all other tests whenever changes are made. With the introduction of the JUnit framework, this automated testing has become much more convenient and effective.² An example test class could look like:

```
public class TestMatrix extends junit.framework.TestCase {
    public void testRank(){
        int rank = Umatrix.rank(new double[0][0]);
        assertEquals(rank, 0);
        rank = Umatrix.rank(new double[][]{{0, 1}});
        assertEquals(rank, 1);
        rank = Umatrix.rank(new double[][]{{1, 2},{3, 4}});
        assertEquals(rank, 2);
    }
}
```

where `TestCase` is part of the mentioned JUnit framework.

²The URL for the framework JUnit is junit.sourceforge.net.

In the `testRank` method, the result of a `rank` routine is checked for the several inputs including special cases, like an empty matrix. It would also be possible to add more methods starting with `test`, which could contain assertions for other functions. The unit testing framework can then be used to automatically run all those tests and report whenever an assertion failed. It is part of all modern IDEs and running tests is typically only a matter of a single mouse click. An assertion failure indicates that the expected state differs from the actual state, which is either an error in the test itself or in the tested class. One should always fix those errors immediately. If an error is detected that is not checked with a test, for example it was reported by a user, it should be fixed and a test case should be set up. This way, the error is guaranteed not to be introduced again. The reason for the success of this strategy of permanent testing is that it is simple and quick enough to set up and maintain even large numbers of test cases, which makes testing an integral part of the development process and is not secretly avoided instead. Last but not least, testing can be fun, because it gives developers much more confidence in the created code. It is also a way for customers to validate certain features of a software product.

However, the design of the classes has to support the task of executing these unit tests. Although a number of solutions exist to run automated tests for GUI components by simulating user behaviour, it is inherently tedious to set up these tests. Therefore, most GUI components are still tested manually, which is always time consuming and error-prone. This is another reason to separate the code for scientific algorithms and graphical user interfaces. The algorithms can only be tested automatically in a reasonably efficient way if they can be called independently.

The scientific procedures in applications based on JStatCom must be automatically testable, because complex algorithms have to be checked against a number of different inputs. Algorithms are often changed to meet certain criteria. Due to the inherent complexity this is a constant source of errors. Automated unit testing can greatly help to discover errors that break existing functionality, although it cannot prove procedures to be correct. However, a reasonable choice of test cases, made up by someone who has a deep domain specific knowledge, is often an excellent guard against programming errors.

The use of testing to validate functionality of software might seem very vague, because first it depends on well-chosen test cases, and second it does not guarantee that a specific function does not fail under special circumstances. A rigorous solution would be to prove the software is correct. This requires to represent the modules under investigation in a formal description language, which can then be verified with an automated theorem prover. A promising project, increasing the feasibility of these procedures with respect to programs written in Java is Bali (Oheimb (2001)). By using this methodology it would be possible to prove certain features of a software to be correct. However, the used methods are highly formalized and still require a considerable effort by experts with knowledge in mechanical theorem proving. They are not likely to be adopted for applications in the considered domain in the near future.

3.5 JStatCom Architecture View Template

This section describes the structure of the presented documentation. Each JStatCom view is presented as a number of related view packets. A view packet is a small, relatively self-contained bundle of information about a particular part of the system or the system as a whole. Two view packets are related to each other as parent/child if one shows a refinement of the information of the other.

This documentation follows a standard template to present each view in a homogeneous form. The views are organized in sections with the following subsections:

1. *Primary Presentation*: Shows the main elements and their relationships that belong to the view packet. It should depict the important aspects of the respective view.
2. *Element Catalog*: Contains a detailed list of all elements that belong to the presented subsystem together with their responsibilities and the corresponding implementation units.
3. *Context*: Presents how the system depicted in the view packet is related to its environment. This can be the interaction with users or clients, but also

with other subsystems of JStatCom. This section only appears if the context for a subsystem is further refined compared to the context of the parent.

4. *Architecture Background*: Explains how important design decisions came to be and reflects on potential alternatives and why they were rejected. The reasons recorded here can be used to justify the current design. This section can also contain further in-depth information on the features of the system under consideration.
5. *Usage Example*: Provides the user with a demonstration that makes explicit use of the respective subsystem. This section only appears if appropriate and if it helps to better understand the system. Most often, examples will be code fragments written in the Java programming language.
6. *Related View Packets*: This section will name other view packets that are related to the one being described in a parent/child capacity.

3.6 View Packet 1: JStatCom

This is the top level view into the framework. It gives an overview of the system as a whole. All subsequent views can be considered as children of this view that split the framework into smaller, intellectually manageable pieces that will itself be recursively refined.

3.6.1 Primary Presentation

The primary presentation shows, how JStatCom is decomposed into five different subsystems. A subsystem is a portion of the system that can meaningfully be considered separately from other parts. It must exhibit some coherent, useful functionality (Clements et al. (2003, p. 62)).

General Resources needed by JStatCom

The framework uses a number of resources that are either required at runtime, for documentation purposes, for testing, or to create deployable software ver-

System	Segment
JStatCom	Data Model
	Components
	Input/Output
	Time Series
	Engine

Table 3.2: Primary Presentation of JStatCom

sions. Table 3.3 lists the resources that are not specific to any subsystem, but are needed for general tasks. The child view packets describe further resources that are needed to run a distinct subsystem. If the respective subsystem is not used, those resources can also be deleted and do not have to be shipped with applications that are based on JStatCom.

Resource	Usage
files in working directory of JStatCom	
doc/	Directory contains the Java API documentation of all classes that are part of the framework. Is a set of HTML files.
jars/	3rd party Java libraries that are required to run JStatCom are stored in this subdirectory. The contained JAR archives must be put in the classpath for compiling and running applications based on JStatCom.
jstatcom.jar	The JStatCom JAR archive. Contains all classes, the default help system, and images. Must be in the classpath for compiling and running applications based on JStatCom.

<code>build.xml</code>	Default build file that can be used with the ANT tool to create a deployable application. It compiles all Java sources in the <code>src</code> subdirectory and creates a JAR archive with the name <code>app.jar</code> . It provides a reasonable default, but should be adjusted for each specific application.
<code>manifest.mf</code>	The default manifest file that is put in the JAR archive of the application to be deployed. It contains information about the classpath and the main class that invokes the application. Should be adjusted for each specific application.
<code>testdata/</code>	Directory that holds several files that are used by unit tests for checking data import functions.
<code>unittests/</code>	Directory with batch files that invoke various unit tests to check the functionality of JStatCom. Tests for special engine implementations are separated and should only be executed, if the requirements for running that engine are met and if this engine is applied. For running the tests for the Gauss engine, a Gauss installation is necessary, for example.

Table 3.3: Resources for MatLab engine, Windows

3.6.2 Element Catalog

Table 3.4 contains the elements of the framework with their name, the implementation unit and their responsibility. Most elements contain further subsystems, which are denoted by bold face entries. All modules are visible across the entire system.

Element Name	Element Responsibility
Implementation Units (com.jstatcom)	
Data Model model model.control util parser	Contains the Type System to define domain-specific data types and the Data Event System to inform listeners about changes in a data object. The Symbol Management is used to share data objects across different components and the Symbol Event System can be used to notify listeners about value changes in a symbol. The Symbol Control provides graphical components to access the state of the symbol manager.
Input/Output io util	Contains classes to support file handling and the Data Import System . It also provides a logging facility.
Time Series ts parser util	This module collects all classes that are especially designed for time series analysis. There are types to represent dates, date ranges and series. It contains the subsystems List , Selection , Table and Calculator for specific tasks.
Components component table equation	This module provides the GUI components that can be used to display and edit data objects as well as to gather user information. The Data Table subsystem contains configurable tables for number arrays and string arrays. The top level application frame is provided in the Application subsystem, and the Equation system is used to display GUI objects for models in matrix notation.

Engine	Contains the abstract engine communications system that hides engine specific implementation details from clients. Subsystems implement the abstract scheme for concrete engines: Gauss , GRTE , MatLab , Stub and Ox . It also has the PCall system for procedure calls.
engine	
engine.gauss	
engine.grte	
engine.stub	
engine.mlab	
engine.ox	

Table 3.4: Elements of JStatCom

3.6.3 Context

Figure 3.1 shows the context of the framework together with the roles that potential users can have. Typically there is someone with domain specific knowledge, who is called *Scientist*, and somebody who develops the Java GUI with JStatCom, called the *GUI Developer*. Only the latter person must interact with the framework. The scientist needs to communicate closely with the developer to lay out the requirements and to set up tests for the software. The GUI developer can focus on the Java side, taking the algorithms as given. User testing should be done mainly by the scientist to make sure that the software meets the requirements. JStatCom serves as an architectural layer that handles all tasks that are common to applications in the given problem domain. Thus it supports the Java developer in incorporating the procedures quickly, in laying out the GUI with specialized components, setting up a help system and managing sets of automated unit tests.

The advantage of this approach is that it is possible, but not necessary, to divide the responsibilities for developing software for analysis modules. This is helpful because this way the GUI programming could be delegated to somebody who does not need to have a deep domain specific knowledge, because she can rely on the algorithm implementations that are already available. Scientists who want to promote their methods by developing easy-to-use programs for potential users could work together with people who have a background in Java programming. JStatCom greatly facilitates this interaction. Other approaches typically

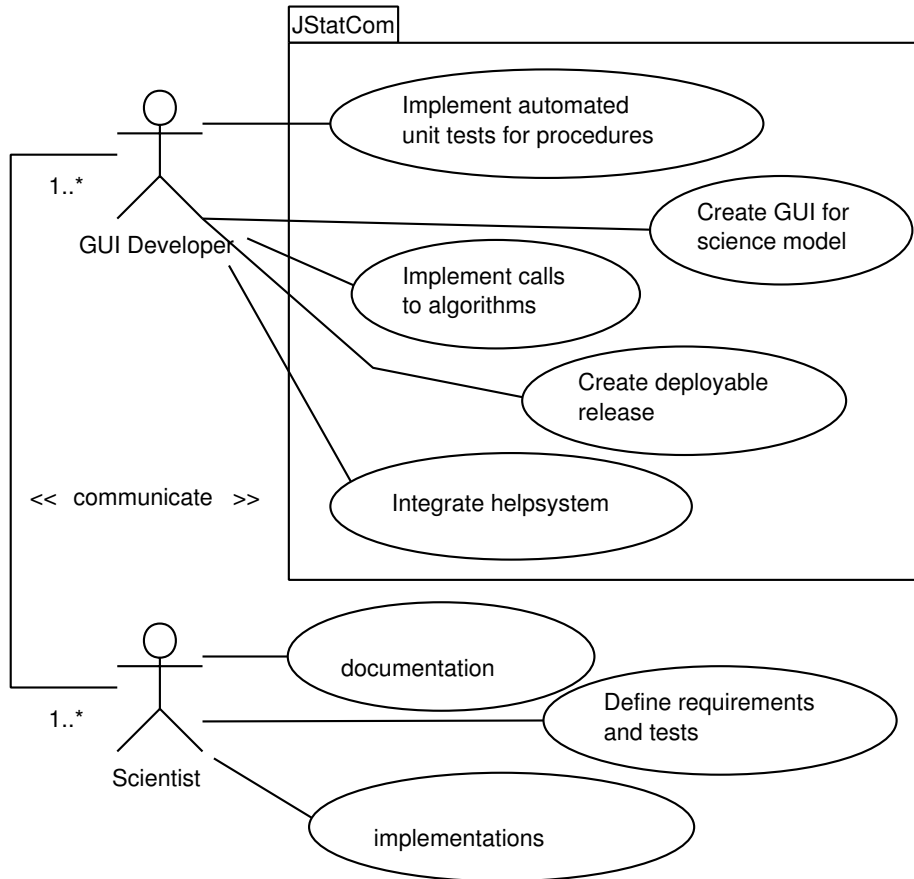


Figure 3.1: Use cases for JStatCom

rely on developers playing both roles at the same time. It is argued that noticeable efficiency gains can be achieved by splitting the development process in the described way, because the participants in a project can do what they know best.

How this scenario works in practise can only be evaluated from experiences with the development of modules for the application JMulTi, because so far no other programs have been created with the help of JStatCom. However, the rapid development of JMulTi would never have been possible without this interaction between various scientists and the GUI programmer. This is proven by the fact that major parts of the involved GAUSS algorithms were in fact programmed by

researchers independently of the Java development. It is also obvious that without JStatCom many of the existing modules now available in JMulti would never have been created because none of the participating scientists could have done this alone with a reasonable effort. Experience has shown that scientists play an important role not only in providing the algorithms, but also in testing the resulting GUI, and in providing documentation which can be used in a help system. It is hoped that JStatCom will be used for other projects as well, especially when several methods should be combined and complex user interfaces are needed.

3.6.4 Architecture Background

During ongoing work with JStatCom, several distinct responsibilities have been identified, which were then organized into different subsystems. These responsibilities broadly correspond to the main requirements for the architecture described in Section 3.4. Although these subsystems are not independent of each other, they can be maintained and extended separately, because they focus on different aspects of the system. Furthermore, all identified segments have a coherent internal structure that separates them from the other subsystems.

3.6.5 Related View Packets

- Parent: View Packet 1: JStatCom

- Children:
 - View Packet 2: Data Model

3.7 View Packet 2: Data Model

3.7.1 Primary Presentation

Table 3.5 shows the five subsystems that the Data Model consists of.

System	Segment
Data Model	Type System
	Data Event System
	Symbol Management
	Symbol Event System
	Symbol Control

Table 3.5: Primary Presentation of Data Model

3.7.2 Element Catalog

Table 3.6 describes the responsibilities of the elements of the Data Model. The Type System, Data Event System, Symbol Management and the Symbol Event System are implemented in the same Java package.

Element Name Implementation Units (com.jstatcom)	Element Responsibility
Type System model util	Contains classes to represent the used data types. An event model is implemented to notify listeners about changes in data values.
Data Event System model	Manages event notification to listeners about changes in data objects.
Symbol Management model	Manages sharing of data objects across different components in a standardized way. Represents data as variables in a collection.
Symbol Event System model	Manages event notification to listeners about changes in symbols.

Symbol Control <code>model.control</code>	Provides components to access all data objects that are shared via the Symbol Management .
--	---

Table 3.6: Elements of the Data Model

3.7.3 Context

Figure 3.2 shows the use cases for the Data Model. The actors are the other subsystems that use the Data Model to carry out several tasks. It can be seen that all other subsystems rely on the services provided by the Data Model.

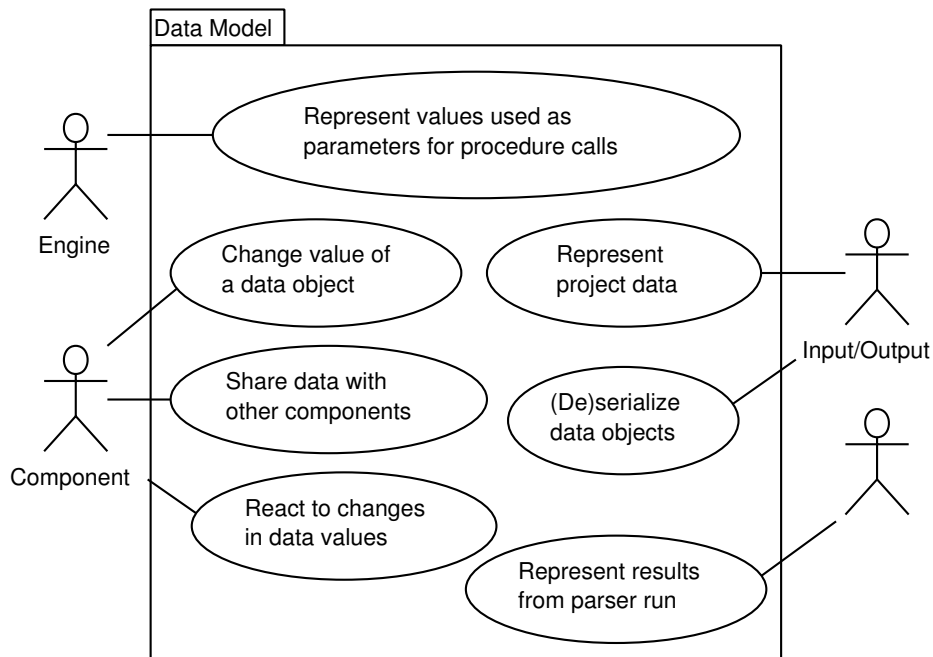


Figure 3.2: Context of the Data Model

3.7.4 Architecture Background

JStatCom needs to represent data internally, because it maintains inputs and results of mathematical computations. Furthermore, it must be easy to let data objects interact with GUI components that display or change the underlying values. The data objects that are used within JStatCom on the Java side must conform to the types that are used by a specific engine. The idea is to have a consistent data management system within the framework that can contain various different types to adjust to any potential modelling situation. When external procedures are called, those types must be converted to and from the respective types of the engine. Another requirement was to let the user of the system access the state of the variables that are used internally. This can be helpful for debugging, but also for making it possible to inspect intermediate results, or values that are not part of the default output. It can also be used to export this data to files. The three subsystems address those issues separately.

3.7.5 Related View Packets

- Parent: View Packet 1: JStatCom
- Children:
 - View Packet 3: Type System
 - View Packet 4: Data Event System
 - View Packet 5: Symbol Management
 - View Packet 6: Symbol Event System
 - View Packet 7: Symbol Control

3.8 View Packet 3: Type System

3.8.1 Primary Presentation

The Java language offers various ways to represent data of different types, for example matrices could be stored in double arrays. However, for JStatCom the core

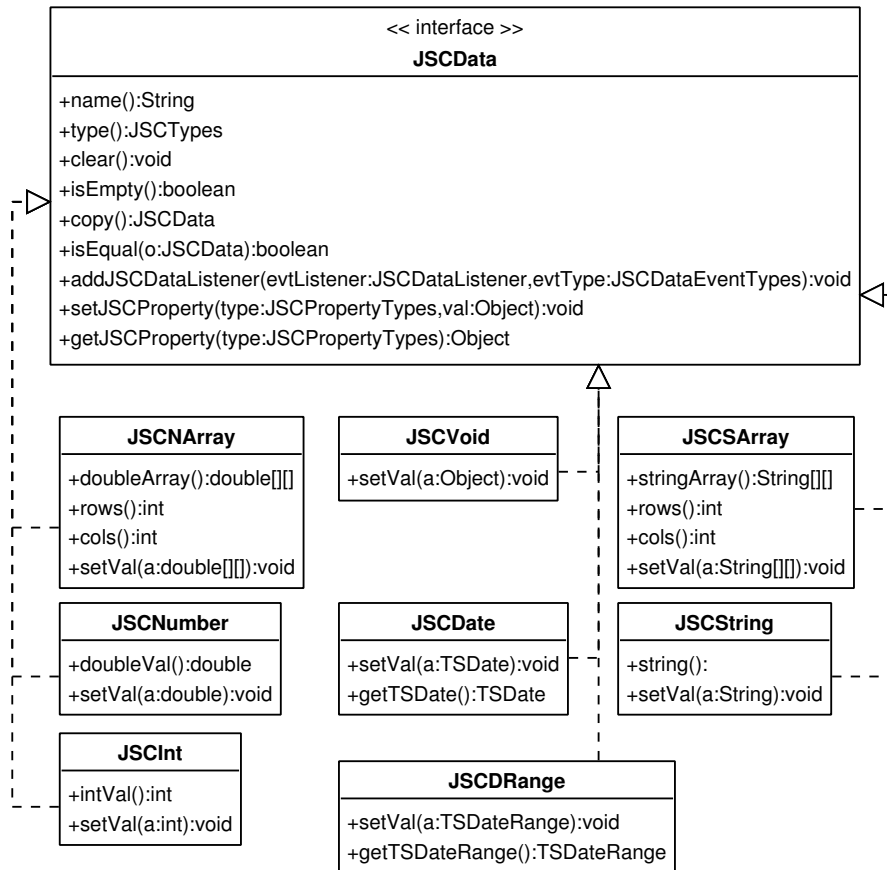


Figure 3.3: Type System

classes and types that the language provides are not sufficient, because they lack a number of features. The Type System must be extendable to represent types that are special to a certain problem domain, for example dates. Furthermore, it should be possible to store each data object permanently to an XML file and retrieve it from there, a feature which is called serialization. Another important feature would be to make all type classes thread-safe. This is a technical requirement for multi-threaded applications, otherwise the state of type instances could become unpredictable when they are accessed by multiple threads. To facilitate the interaction between the Data Model and GUI components, the Type System

must also provide event-based notification about changes in the underlying data.

JStatCom uses a metadata model to achieve a high degree of flexibility. Core attributes are standardized for all data types by defining a very general interface `JSCData`, which all specific types must implement. This interface only specifies methods that are common to all potential types. Any specialized functions to access or modify the contents of data objects are defined in implementations of the interface. Type related code and interfaces are therefore strictly separated.

Figure 3.3 shows the complete interface and all types that are currently implemented. For the sake of clarity, only very few methods of the actual data classes are given, a complete documentation can be found in the API documentation. One of the advantages of having an interface `JSCData` that all types implement is that it is possible to store instances of different types in a single array `JSCData[]`. This can greatly simplify method signatures and it is used by the Engine system, see Section 3.13, to provide a standardized `Engine.call` method that takes two arbitrary data arrays as parameters for input and output. Each engine can then check the elements of these arrays for their type via `JSCData.type` and handle them accordingly, a mechanism that is completely hidden from engine clients.

Variable Names

In JStatCom every data object has a name. As a general convention throughout the framework, variable names can contain letters, numbers and `'_'` but must start with a non number. To illustrate, `_invest`, `invest` and `i2` would be valid names, whereas `2i`, `gov exp` or `cons+inv` would be invalid. This is enforced wherever variable names can be set within the program. Variable names are case insensitive, meaning that two variable names which differ only in the case of one or more letters are considered equal.

3.8.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.model</code>	
JSCData	Interface that defines the general contract for all implemented types.
JSCTypes	Enumeration of all data types.
JSCPropertyTypes	Abstract class to be subclassed if special properties for data classes should be defined.
JSCConstants	Helper class that defines commonly used constants. It is also used to enforce the variable names conventions.
JSCDate	Implementation of JSCData that represents a date type.
JSCDRange	Implementation of JSCData that represents a range defined by two dates.
JSCSArray	Implementation of JSCData that represents an $n \times m$ array of strings.
JSCString	Implementation of JSCData that represents a single string.
JSCNArray	Implementation of JSCData that represents an $n \times m$ array of double numbers.
JSCNumber	Implementation of JSCData that represents a double number.
JSCInt	Implementation of JSCData that represents an integer number.

JSCVoid	Implementation of JSCData that represents a reference to an arbitrary object.
NumberRange	Represents an interval defined by two real numbers. It does not have a corresponding type yet.
NumberRangeTypes	Enumeration of all possible interval types.
<code>com.jstatcom.util</code>	
NumberRangeParser	Parser that assembles number range objects from an input string.
<code>com.jstatcom.util</code>	
UMath	Holds static utility methods for special math operations.
UMatrix	Holds static utility methods for various simple matrix operations.
UString	Holds static utility methods for string operations.
UStringArray	Holds static utility methods for various simple string array operations.

Table 3.7: Elements of the Type System

3.8.3 Architecture Background

The implemented data types are mainly responsible to facilitate interaction with GUI components and to operate as storage units, instead of carrying out computations on them directly. For example, the JSCNArray class is a basic matrix class for JStatCom, but it does not try to compete with existing Java matrix implementations for linear algebra calculations. The benefit is that the interfaces of all types are kept quite simple. However, data can easily be moved from JSCData types to instances of specialized math classes. But typically sophisticated linear

algebra calculations are done with the employed external engine, which is especially suited and optimized for that purpose. Another effect of the taken solution is that instances of `JSCData` cannot change their type anymore after they have been created. This introduces a form of type-safety for data objects within the framework.

An alternative to the chosen metadata model would have been to use one general `VALUE` class that can take on different states, depending on what type of data is stored. This has the advantage that `VALUE` instances could always be treated uniformly, but it tends to create a monolithic class with many unrelated functions for different data types. The presented approach still offers the possibility to treat `JSCData` instances uniformly, but only with respect to their interface, which is quite general. However, the benefits clearly outweigh this drawback, especially because this approach allows to have an arbitrarily rich type system.

An advantage of the taken solution is that the system can be extended with arbitrary new types in a very straightforward manner without interfering with existing types by just creating new realizations of `JSCData`. However, defining a new type for the core framework is not a trivial task, because the new class should be thread-safe, it must inform listeners about changes in the data, it must be XML serializable and it must be well-documented and tested. If necessary, there should also be GUI components to access and modify the contents of a type. Future enhancements of `JStatCom` could include types of complex numbers and arrays, or types for arbitrary precision numbers and big integers. Even multi-dimensional arrays could be considered.

Another special feature of the Type System is that every data object must have a name. This convention was chosen, because it helps to identify variables during runtime. Especially when error messages are created, it is often extremely useful to have the name of the variable that was involved. Each instance of `JSCData` should be viewed as a named storage container.

A requirement for the Data Model was that it can be used in a multi-threaded environment, because it is often desirable to let computationally intensive procedures run in their own thread, which might lead to the situation where data objects are modified from different threads. This has implications for all methods that access mutable data. For an exhaustive discussion of the consequences, especially

for thread-safe programming with Java, see Lea (2000). Therefore all access to mutable data is synchronized and data objects can be used from multiple threads safely.

Lastly, it was important to provide means to store and restore JSCData instances to and from files. This is done via the XML serialization mechanism provided by the JADE toolset. The feature could later be used by a project management system to store model settings permanently in a standardized way.

3.8.4 Usage Example

This example creates instances of three different types, JSCNArray, JSCDate and JSCInt. The variable `y` could, for example, represent some data for a univariate AR model the variable `start` could be the starting date, and `lags` could be the number of lags. Although these three variables represent different data types, they can be put together in an array of type JSCData, because they all implement this interface. It is often very helpful to treat different data types uniformly as JSCData, for example, the `Engine.call(String procName, args JSCData[], retArgs JSCData[])` method takes data arrays as arguments for input and return parameters. However, if more specific functionality than defined in the JSCData interface is needed, a cast to the specific type is necessary afterwards.

```
// data instances of various types are created
JSCNArray y = new JSCNArray("yData",
                           new double[]{2.3, 1.9, 3.3, 5.5, 3.4});
JSCDate start = new JSCDate("start", new TSDate(1960, 1, 4));
JSCInt lags = new JSCInt("p", 1);

// all data can be treated uniformly as JSCData
JSCData[] args = new JSCData[]{y, start, lags};
...
// if the concrete implementation is needed, casting is necessary
// but the type can be checked before
JSCTypes type = args[0].type();
```

```

if (type == JSCTypes.NARRAY){
    JSCNArray yRef = (JSCNArray) args[0];
    // prints the value 2.3 to the standard output
    System.out.println(yRef.doubleAt(0,0));
}

```

3.8.5 Related View Packets

- Parent: View Packet 2: Data Model
- Children: none

3.9 View Packet 4: Data Event System

3.9.1 Primary Presentation

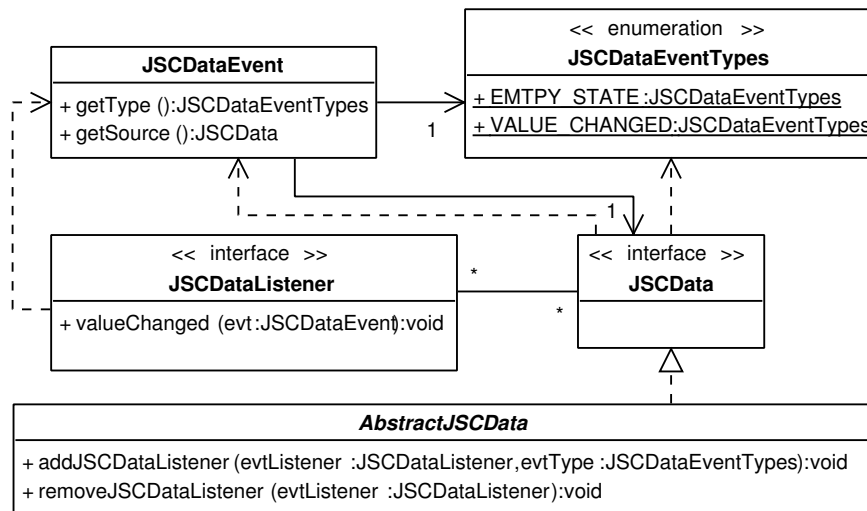


Figure 3.4: Classes in Data Event System

The Data Event System is used to notify listener objects about changes in JSCData instances. Listeners must implement the JSCDataListener interface.

If a change in the data value of a data object occurs, a `JSCDataEvent` of a certain type is created and the method `JSCDataListener.valueChanged` of the listener interface of all registered listener objects is called with the event parameter. The listener can then inspect this event and invoke some action. The types that events can have are defined in the enumeration class `JSCDataEventTypes`. The types `EMPTY_STATE` and `VALUE_CHANGED` are available. The first event type should be used by `JSCData` implementations when a data object changed its state from empty to not empty or vice versa. The second event should be used whenever value changes occur, including changes in the empty state. The rationale behind this is given in Section 3.9.3. An abstract default implementation of the `JSCData` interface that manages adding and removing of listeners is in `AbstractJSCData`. Concrete `JSCData` implementations should therefore inherit from `AbstractJSCData` to reuse this default functionality.

3.9.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.model</code>	
<code>JSCData</code>	Interface that defines the general contract for all implemented types.
<code>JSCDataListener</code>	Interface to be implemented by listeners to events fired by data objects.
<code>JSCDataEvent</code>	Events of this class are used to notify <code>JSCDataListener</code> objects.
<code>JSCDataEventTypes</code>	Enumeration of all possible types that a <code>JSCDataEvent</code> can have.
<code>JSCDataEventSupport</code>	Support class for listener management.

<code>AbstractJSCData</code>	Abstract implementation of the <code>JSCData</code> interface which provides default functionality for handling of the different listeners.
------------------------------	---

Table 3.8: Elements of the Data Event System

3.9.3 Architecture Background

The Data Event System is an essential part of the JStatCom Data Model, because it provides a way to inform objects about changes in data values. It is an example of the classical *Observer* pattern. The listeners just need to register with a data object and do not need to know anything about each other, nor where the change in the data value actually was made from. Thus, objects can communicate via events and are further decoupled. If another component should be added to or removed from the list of listeners to a certain data object, this would not affect the other listeners at all. An event driven data model is especially helpful to use data values in conjunction with GUI components, which need to update their state according to changes in the underlying data. However, this is especially addressed by the Symbol Event System, described in Section 3.11, but the Data Event System is the prerequisite for it.

A common problem with event driven systems is that inefficient updates might occur. This happens when listeners are informed about changes that are not relevant for them. Usually these updates are just ignored, because listeners should check whether the condition for taking an action is met. But it can have serious performance implications, either if there are many listeners that have to be informed, or if wrong updates occur very often, or if it is costly to generate notifications. For the problem domain of JStatCom it might occur that listeners are informed when the value of a data object changed, but that the action is only taken when the object changed its state from empty to not empty or vice versa. Other changes are ignored. This can lead to inefficiency, because listeners receive updates too often and, furthermore, it can also be expensive to generate those updates.

A common solution to this problem is to define specific event types that the listeners can register for. For this reason the Data Event System defines the `JSCDataEventTypes` class, which holds the two event types `EMPTY_STATE` and `VALUE_CHANGED`. If listeners just register to `EMPTY_STATE` events, then they are only informed when the empty state of the data object changes, but not if the value changes to some other value.

This has another effect which may improve performance even more. It can be quite costly for data objects to generate `VALUE_CHANGED` events, because those events are only generated if a true value change has actually occurred. For some data types, like potentially large number arrays, it can be necessary to compare every single element to make sure that no change has occurred, like for example when `JSCNArray.setValue(double[][] arg)` is called. But these data objects only need to check this, if there are any listeners registered to the `VALUE_CHANGED` event type. Otherwise this potentially costly check is simply skipped. If only `EMPTY_STATE` listeners are registered, the only thing that needs to be checked is whether the object changes its empty state, which is always a cheap operation. Thus, by defining different types of events and letting listeners receive only the relevant updates, the efficiency of the event handling can be improved, because fewer updates are done and time consuming checks may be skipped. Implementations of the `JSCData` interface should make use of the features provided by the Data Event System, especially if they become part of the core framework.

3.9.4 Usage Example

The code example demonstrates the usage of the Data Event System. However, if the action that should be taken on a change in the data value is somehow related to a graphical user interface component, one should rather use the Symbol Event System to install listeners. The reason for this will be explained in Section 3.11.

Here a `JSCNArray` is instantiated that could hold the residuals of an OLS regression, for example. Then a listener object `resListener` is created by an inner class that implements the interface `JSCDataListener`. The method `valueChanged` defines, what action is going to be taken, if the underlying data object is either empty or not empty. This can easily be inspected via the

`isSourceEmpty` method of `JSCDataEvent`. It can be seen that the listener only needs to be informed when the empty state of the residual data object changes. Therefore it registers with the parameter `JSCDataEventTypes.EMPTY_STATE`. This has the advantage that updates occur only when the empty state changes, and that the residual object does not need to compare values if it has been set to just some other value without changing its empty state. The last line of the code example invokes `clear` on the data object, which would inform `resListener` that `residuals` has changed from not empty to empty, because all data has been deleted. It should be noted that it does not matter for `resListener`, where the `clear` method is invoked from.

One could also install other listeners that take different actions, without affecting already existing listeners. This shows that event based communication greatly helps to reduce direct dependencies among classes, but it can also be seen that it can become difficult to predict all effects of a change in some variable if there are many listeners in various different code locations. Therefore, in more complex applications, one should think of organizing listeners in a single *Mediator* class that is especially dedicated to that purpose.

```
// array holding residuals from an estimation
JSCNArray residuals = new JSCNArray("res",
    new double[]{2.3, 1.9, -3.3, 5.5, -3.4});
// create a listener object
JSCDataListener resListener = new JSCDataListener(){
    public void valueChanged(JSCDataEvent evt){
        if (evt.isSourceEmpty())
            // do some action, because source was empty
        else
            // do some action, because source was not empty
    }
};
// register listener to residuals
residuals.addJSCDataListener(resListener,
    JSCDataEventTypes.EMPTY_STATE);
// empty residuals, listener will take action
residuals.clear();
```


3.9.5 Related View Packets

- Parent: View Packet 2: Data Model
- Children: none

3.10 View Packet 5: Symbol Management

3.10.1 Primary Presentation

The Symbol Management system is used to treat data objects that were created with the Type System as variables and to share them across different components in a standardized way. Figure 3.5 gives an overview. It consists of the class `SymbolTable`, which is an aggregation of an arbitrary number of `Symbol` instances. Each symbol object represents exactly one instance of `JSCData`. Symbol objects are identified via their name in a symbol table, which operates as a shared data repository. Via the symbol table it is possible to access the symbol elements and finally the actual data values. Symbols can be understood as pointers to data objects. The referenced values, instances of `JSCData`, can be changed efficiently during runtime. A symbol can hold any data type on creation, but afterwards it is not possible to change the type anymore. For example, if a symbol was initialized to point to a `JSCInt`, then a runtime exception would be generated when trying to set it to a `JSCString`. This introduces a form of type-safety to the Symbol Management.

`JStatCom` offers a way to limit the visibility of symbol tables to only components that belong to one model. Furthermore, it is possible to share data on different levels, which is somewhat similar to global and local variables. For this, the interface `SymbolScope` is provided. Implementations of this interface have access to symbol tables on three different levels: global, upper and local. Every symbol table keeps a reference to the next higher symbol table in the hierarchy defined by implementations of `SymbolScope`. The top level symbol table has only a `null` reference instead. This mechanism can be very helpful to organise shared data within an analysis module. Data that is global can be accessed from all child

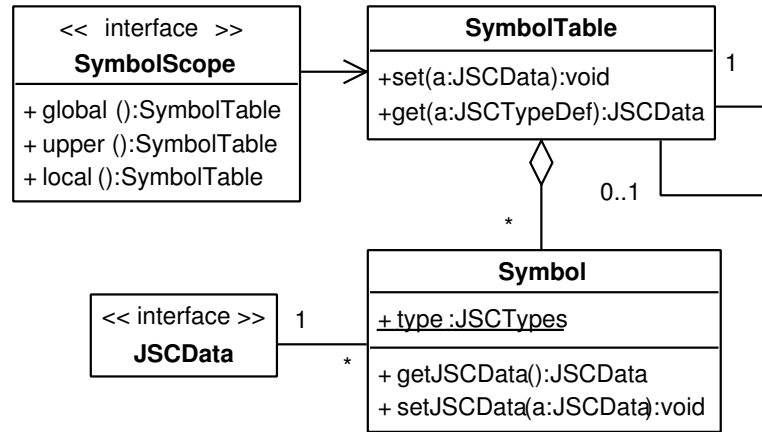


Figure 3.5: Symbol Management

components that implement the `SymbolScope` interface via a call to `global`.³ But sometimes also finer grained sharing can be helpful. A class can use a local symbol table to share data in a symbol table that is accessible via `local` from within itself, and `upper` from within its child components. Thus it can share variables on a lower level that is not visible to components that are parents or siblings in the component hierarchy, thus avoiding to put too many variables in the global table. This feature is especially useful in complex systems with many shared variables.

To be more specific, Figure 3.6 shows, how the `SymbolScope` interface is implemented by components of the model. It should be noted that the implementing components must make sure that the correct symbol tables are referenced. Developers can rely on the two default implementations `ModelFrame` and `ModelPanel` which should be used as superclasses for all panels that need access to shared variables. These classes make sure that the correct symbol tables are referenced according to the component hierarchy imposed by the GUI design.

³The terms child and parent are defined here in terms of the component hierarchy of GUI classes in Java Swing. Two components have a parent/child relation if the parent component contains the child component. For example, if a panel `ResAnalysis` contains the panels `DiagPanel` and `SpectrumPanel` than these two panels are called *children* of `ResAnalysis` which is the *parent*.

Every model should therefore be implemented with a `ModelFrame` as the top level component. This can be the starting point for any application based on `JS-tatCom`. A `ModelFrame` is typically a composition of a number of `ModelPanel` components. Both classes provide access to the Symbol Management system and can use it to set and retrieve variables. The `SymbolScope` interface imposes a hierarchical ordering of symbol tables. The `ModelFrame` and `ModelPanel` implementations of this interface use the GUI component hierarchy for this. Symbol tables are assigned as follows:

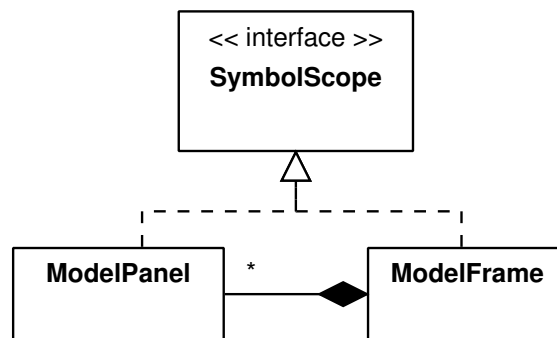


Figure 3.6: `SymbolScope` inheritance

- `ModelFrame` - top level component, `global`, `local` and `upper` are equivalent and return the top level symbol table
- `ModelPanel`
 - `local` - returns the symbol table created by this panel
 - `upper` - searches the component hierarchy upwards until an instance of `SymbolScope` is found and returns the result of a call to `local` on the component found; if no parent instance of `SymbolScope` exists, `this.local` is used
 - `global` - searches the component hierarchy upwards until an instance of `SymbolScope` is found and returns the result of a call to `global`

on the component found; if this instance is a `ModelPanel`, it will search itself for the next higher component, and so on, typically the global table defined in `ModelFrame` is reached; if no parent instance of `SymbolScope` exists, `this.local` is used

It should be noted that this process is done automatically. Developers should only understand that `ModelPanels` can be used to define access scopes. One could also think of other possible implementations of `SymbolScope`, reflecting different hierarchical schemes.

3.10.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.model</code>	
<code>Symbol</code>	Represents a variable with a certain type and name.
<code>SymbolScope</code>	An interface to be implemented by components that provide access to a hierarchy of symbol tables. Used by <code>ModelFrame</code> and <code>ModelPanel</code> .
<code>SymbolTable</code>	Contains a sorted collection of <code>Symbol</code> objects that can easily be accessed.
<code>JSTypeDef</code>	Defines a variable with a name, a type and an optional description. Can be used to retrieve symbols from a symbol table.
<code>Scope</code>	Type-safe enumeration containing all existing scope types, namely global, local and upper.
<code>ModelFrame</code>	An internal frame that can be used as top level container for modules in the <code>JStatCom</code> framework. It holds the global symbol table that can be accessed by all child components.

ModelPanel	A panel to be used by components that need access to the symbol table hierarchy.
------------	--

Table 3.9: Elements of the Symbol Management

3.10.3 Architecture Background

The Type System introduces various ways to store and manipulate data of different kind. However, a common problem when designing applications for complex models is that various classes and GUI components need to share data stored in instances of JSCData. For example, when a VAR model is analysed, then there are variables that define the state of the model, like lags, subset restrictions, data for endogenous, exogenous and deterministic variables, and so on. The user interface is typically broken up into several components that handle different modelling steps, like specification, estimation, diagnostics, and forecasting. All these components need to have access to the model state. It would certainly not be a good idea to exchange data directly between these components, because this would create unnecessary dependencies among them. Another anti pattern is of course to rely on global data, because this would break data encapsulation, one of the principles of object-oriented programming. More generally speaking, by referencing variables directly one would also unnecessarily create dependencies between classes that could be avoided by using the Symbol Management system. The reason is that if variables would be referenced, a compile-time dependency would be created which establishes a static relation between classes. The Symbol Management system instead only creates references at runtime by inspecting the GUI hierarchy. This is much more flexible because it allows to use one component together with different classes, thus facilitating reuse. The component must only know the names of the symbols to look up in the respective table. The reference to the symbol table will only be established when the GUI is shown.

In the application JMulti this has been done for components that are needed in various different contexts, for example for the panel that computes a kernel density estimation. Each instance of that component references different variables which it retrieves from the respective symbol table via their names.

The `SymbolTable` can represent the state of arbitrary models as an aggregation of symbols of different types. The data representation and the sharing mechanism are standardized and can be reused for any modelling situation. If the available types are not specific enough, user-defined extensions of the Type System could be considered. The Symbol Management system would not need to be changed in any way to handle new types, because it treats variables uniformly as `JSCData`. Therefore, it is quite easy to share model data among GUI components that are part of a certain scientific analysis module.

3.10.4 Usage Example

Figure 3.7 sketches, how classes for a VAR model interface could be laid out with `ModelFrame` and `ModelPanel`. The top level component for the model is `VARFrame` which is composed of a panel for model specification and a panel for residual analysis. The latter is itself composed of a panel for diagnostic tests. Each panel can access the Symbol Management system easily, because it inherits the access methods `local`, `upper`, `global` from `SymbolScope`.

A snapshot of the object structure at runtime is presented in Figure 3.8. The entities of the diagram are now objects instead of classes. It can be seen that the instance `frame` of the class `VARFrame` has a link to a symbol table `global`. This is usually the place to store variables that should be shared by all panels that a certain model frame is composed of. It cannot be accessed by panels from other model frames, at least not by default. In a VAR context, the global symbol table should contain the selected data and lags, estimated coefficients, standard deviations, names of variables, etc.. Model panels, like `panel1` for specification and `panel2` for residual analysis, have access to the global symbol table via their `global` method. However, a further refinement is that data can also be shared on lower levels. For example, it might be that some data is shared by panels belonging to the residual analysis only, which are children of `ResAnPanel`. Therefore

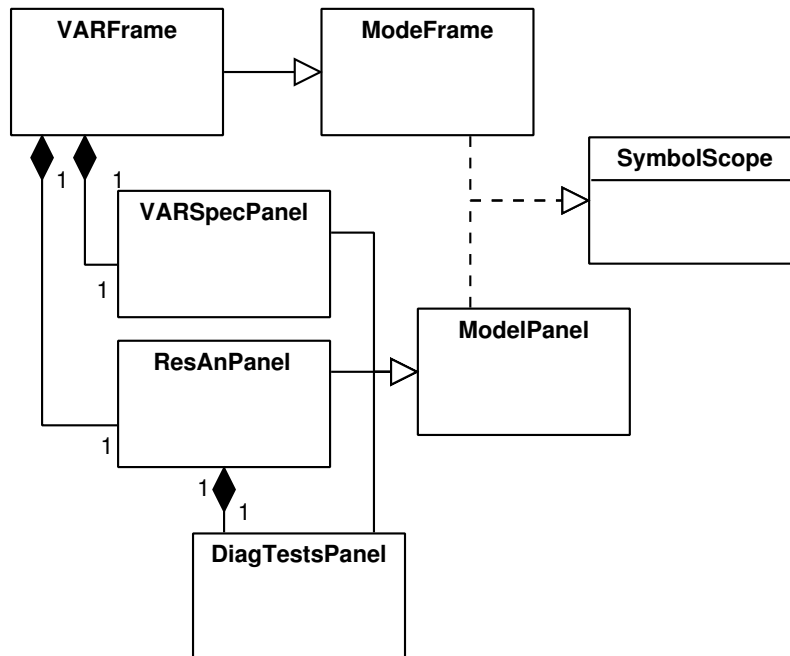


Figure 3.7: Class structure of a hypothetical VAR frame

the respective symbol table `local2` can be accessed via the `upper` method by `panel21`, the object to hold the diagnostic tests interface. But panels might also use a symbol table to store variables that are not used by other components, for example test statistics and p-values of diagnostic tests might go to `local21`. This data need not to be shared, but it might still be reasonable to put it in a local symbol table. However, the local symbol table of a panel is the upper symbol table of child components, thus `local2` can be accessed by `panel21`.

Storing data in symbol tables is not only meaningful when variables should be shared, but it can also be used to publish the results in the Symbol Control system (see Section 3.12), which is another subsystem of JStatCom that provides access to variables that are currently used. It presents a tree view of the symbol table hierarchy and it has components to display and export all symbols that have been put in one of the symbol tables.

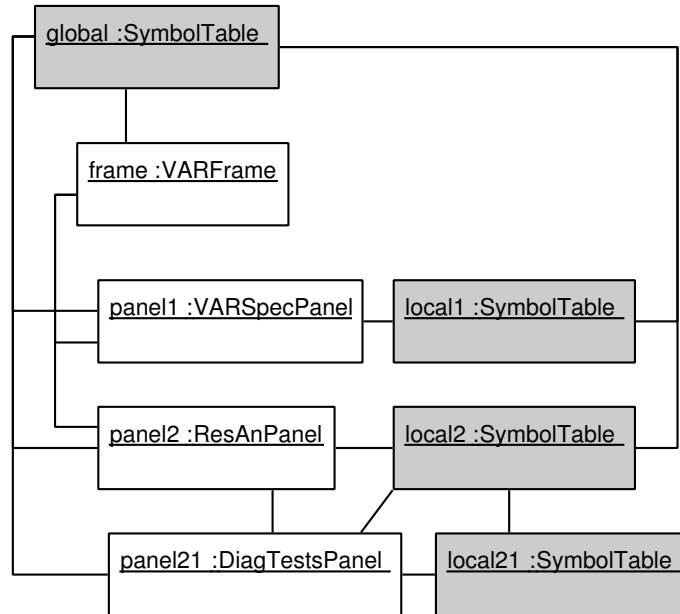


Figure 3.8: Snapshot of model objects and shared data with different scopes

The following small Java code example should demonstrate the workings of the Symbol Management system. It corresponds to the class diagram in Figure 3.7, but only sketches the contents of the concrete implementations. The `VARFrame` binds all panels together and should provide a mechanism to navigate between them. `VARSpecPanel` should contain a mechanism to select series and to specify lags. The Components subsystem provides classes for that purpose. As a placeholder for this, only a `JSCString` with the estimation method is stored globally. The `ResAnPanel` sets the names of the residual series locally in its `setResidNames` method. Thus, they can be accessed by child panels, like `DiagTestsPanel`. The method `DiagTestsPanel.executeTests` invokes the test procedures. The respective input parameters can easily be retrieved by their names from the global and upper symbol tables. The actual tests would typically be invoked via the Engine system, which is described in the next section.

```
// top level class, contains various panels
public class VARFrame extends ModelFrame {
    private ResAnPanel resAnalPanel;
```



```
private VARSpecPanel vARSpecPanel;
...
public VARFrame(){
    super("VARFrame");
    // add menubar or tabbed pane
    // add panels
}
} // end VARFrame
// panel for model specification

public class VARSpecPanel extends ModelPanel {
    ...
    // sets estimation method as JSCString to global table,
    // variable is shared by all ModelPanel children
    private void setEstimationMethod(){
        global().set(new JSCString("EstimationMethod", "OLS"));
    }
} // end VARSpecPanel

// panel for residual analysis, contains ModelPanel children
public class ResAnPanel extends ModelPanel {
    public DiagTestsPanel diagTestsPanel;
    ...
    // constructor
    public ResAnPanel(){
        super();
        // add child panels, maybe with a tabbed pane
    }
    // set the names of the residuals in local table
    // local table is upper table for child ModelPanels
    private void setResidNames(){
        local().set(new JSCArray("ResNames",
            new String[]{"u1", "u2", "u3"}));
    }
} // end ResAnPanel
```

```
// ModelPanel to carry out diagnostic tests
public class DiagTestsPanel extends ModelPanel {
    ...
    // gets estimation method from global table
    // and residual names from upper table
    private void executeTests(){
        JSCString estMeth = global().getSymbol("EstimationMethod")
                                .getJSCString();
        JSCSArray resNames = upper().getSymbol("ResNames").getJSCSArray();
        ... // invoke procedure via Engine system
    }
} // end DiagTestsPanel
```

This code should only give an idea of how the Symbol Management system could be used. It has the advantage that there are fewer direct connections between components. `DiagTestsPanel`, for example, does not know anything about `VARSpecPanel`, although it uses variables that were set by this panel. The code sketch here uses plain strings to define variables. This is suitable only for small applications, because one might easily mix up names, especially if there are many variables. A much better way is to create a separate class with the definitions of all shared variables in a certain scope. The framework supports this with the class `JSCTypeDef`, which can be used to define variables with their name, the type and an optional description. Using this way of defining shared data helps greatly to manage even large GUI systems with many variables. It is part of the design guidelines to build extendible applications with `JStatCom`.

3.10.5 Related View Packets

- Parent: View Packet 2: Data Model
- Children: none

3.11 View Packet 6: Symbol Event System

3.11.1 Primary Presentation

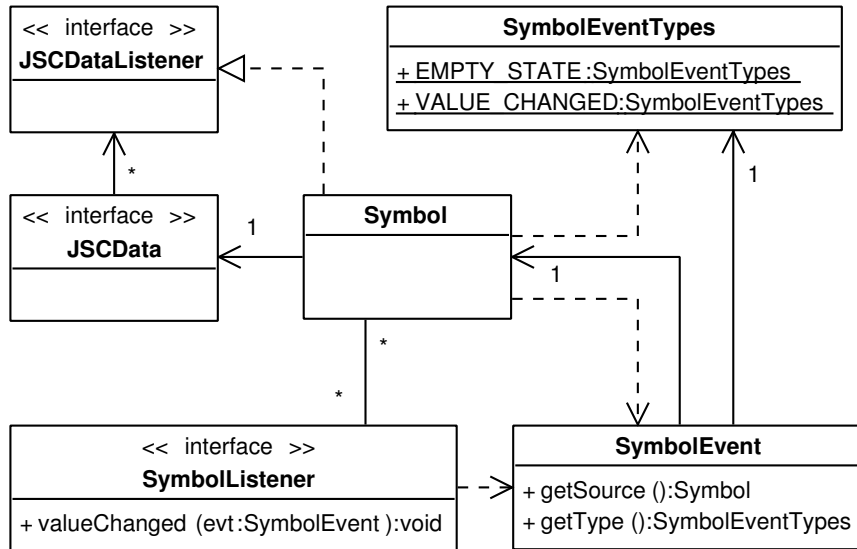


Figure 3.9: Classes in Symbol Event System

The Symbol Event System is used to notify listeners about changes in a **Symbol** object. A symbol changes when the value of the referred **JSCData** object is modified, or if a new **JSCData** object is set with a different value. Symbols implement the **JSCDataListener** interface and are informed about changes in the underlying data object via the Data Event System. The type and the name of a symbol can never change after it has been created. Listeners that want to be informed about symbol changes need to implement the **SymbolListener** interface and they must register with the symbol object. Whenever a change occurs, the `valueChanged` method of the **SymbolListener** interface is called with an event parameter. The event is of type **SymbolEvent** and can be used by the listener to inspect the source and type of the event and the value before and after the change. Typically, listeners retrieve the symbol from a symbol table and register themselves with that object.

All event notifications that are created by instances of `Symbol` are synchronized with the Swing Event Dispatching Thread, unlike events fired by `JSCData` objects. This guarantees that any GUI related activity resulting from a change in a symbol is executed within that thread, even if the change in the symbol was originated from another thread. This is important, because otherwise unpredictable behaviour of the graphical user interface could result. This makes the `Symbol` Event System an adapter between the Data Event System and the GUI, because the underlying computations might run in different threads, but any GUI related methods must run in the single Event Dispatching Thread. For this reason, developers should use the `Symbol` Event System rather than the Data Event System for listeners in graphical applications.

3.11.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.model</code>	
<code>SymbolEvent</code>	Events of this class are used to notify <code>SymbolListener</code> objects.
<code>SymbolEventTypes</code>	Enumeration of all possible types that a <code>SymbolEvent</code> can have.
<code>SymbolListener</code>	Interface to be implemented by listeners to events fired by symbol objects.
<code>SymbolEventSupport</code>	Support class for listener management.

Table 3.10: Elements of the `Symbol` Event System

3.11.3 Architecture Background

On first glance, the Symbol Event System and the Data Event System seem to have the same purpose - notification of interested listeners about changes in data values. However, symbols are not exactly equivalent to data values. They merely keep a reference to some data object, which might change during the lifetime of a symbol. A listener to a symbol is informed, when the referenced data object changes its value, or when the reference changes to point to another data object (of the same type) with a different value. If a listener would register directly with a data object, then it would only be informed about changes in the symbol as long as the symbol references the same object. But typically, listeners need to be informed about changes in shared data values, which are represented by symbols, therefore they should listen to symbols rather than data objects.

Actually, the first version of the Symbol Management did not have the distinction between data objects and symbols. There was no Symbol Event System, only data events. Data objects were stored directly in the symbol table. The drawback was that once a data object was stored, the reference to it could not be changed anymore. This had the effect that it was not possible to just replace the reference of an element in the symbol table to point to a different data object, because all listeners to the first object would have lost the connection to the new element in the symbol table. Instead, the values of the new data object had to be copied to the already existing object, a potentially costly operation. The current system is more efficient, because the actual value and the reference to the symbol is separated between a `JSCData` and a `Symbol` object. The value of a symbol can efficiently be changed by just pointing to a new data object, without additional copying. The only drawback of this system is that event handling becomes rather complex, because listeners can either register directly with data objects, or with symbols.

Another important aspect of the separation between the Data Event System and the Symbol Event System is that sometimes the synchronization with the mentioned Event Dispatching Thread is not desirable in a multi-threaded application. In fact, it can make event notification among different threads impossible. Therefore only the Symbol Event System is synchronized with this thread, but not the Data Event System.

To summarize, although the concept of a symbol and, as a consequence, the introduction of the Symbol Event System seems a bit complicated in the beginning, it solves some subtle problems related to performance and concurrency. Therefore it is an inevitable part of the Data Model in JStatCom. However, the complexity can be reduced for the developer, if the Symbol Event System is used by default and the Data Event System is seen as the mechanism that is only required in the background.

3.11.4 Usage Example

The use of the Symbol Event System is demonstrated with a code example that sketches possible implementations of an estimation panel and a residual analysis panel for a VAR analysis. In the class `EstimatePanel`, a type definition `RES` for the symbol holding the estimated model residuals is created. This should be the preferred way to define variables, although a simple string would also have been possible. But it would lack type information and a description.

In the `estimate` method, the residuals are created and set to the global symbol table. This way, they are shared with all other components that are part of the VAR frame, like the residual analysis. It should be noted that the symbol is retrieved via `global().get(RES)` by using the type definition. This also sets the description of the symbol, which can then be accessed by the Symbol Control system. Especially if there are many variables, symbol descriptions can be an important piece of information for maintaining and using the software.

The residual analysis panel `ResAnPanel` uses the estimated model residuals by accessing the global symbol table together with the type definition `EstimatePanel.RES`.⁴ The code example demonstrates, how a `SymbolListener` is created and attached to that symbol. Updates occur only, when the estimated residuals change their empty state. Typically, the listener would cause the panel to be disabled if there are no residuals available, or activate it otherwise.

⁴In the current Java release 5.0, this would simplify to `RES`, if `EstimatePanel` is statically imported. The source code would become shorter and clearer.

```
// panel for model estimation
public class EstimatePanel extends ModelPanel{
    // type definition for residual symbol, better than just
    // using a string, has name, description and type
    public static final JSCTypeDef RES = new JSCTypeDef("residuals",
        JSCTypes.NARRAY, "estimated VAR residuals");

    // some estimation is done here
    private void estimate(){
        // array holding residuals from an estimation
        // for this example an arbitrary array is created
        JSCNArray residuals = new JSCNArray("res",
            new double[]{2.3, 1.9, -3.3, 5.5, -3.4});
        // gets global symbol table
        // gets symbol defined by RES
        // sets values to estimated residuals (listeners are informed)
        global().get(RES).set(residuals);
    }
}

// residual analysis panel, needs to check
// whether estimation output is there
public class ResAnPanel extends ModelPanel{
    // initializes this panel, attaches listeners
    private void initialize(){
        // symbol listener that does something if the
        // symbol for estimated residuals is empty or not
        SymbolListener listener = new SymbolListener(){
            public void valueChanged(SymbolEvent evt){
                if (evt.isSourceEmpty())
                    // do some action, because source was empty
                else
                    // do some action, because source was not empty
            };
        };
    }
}
```

```
// gets global symbol table
// gets symbol with estimated residuals
// attaches listener to EMPTY_STATE events only
global().get(EstimatePanel.RES).addSymbolListener(listener,
    SymbolEventTypes.EMPTY_STATE);
}
}
```

It can be seen that the Symbol Event System allows to change the state of GUI components according to the state of the variables of a model easily. However, as with the Data Event System, it can be hard to predict the effect of a change in one symbol, if there are many listeners attached. If a complex modelling frame is created, its behaviour could be difficult to understand. A possible solution to this would be to keep all listeners in a special location, for example a single class. It also helps to define listeners always in the same place within classes, for example in the `initialize` method.

3.11.5 Related View Packets

- Parent: View Packet 2: Data Model
- Children: none

3.12 View Packet 7: Symbol Control

3.12.1 Primary Presentation

The Symbol Control System is used to represent the hierarchy of symbol tables together with their associated symbols in a tree structure. This is illustrated in Figure 3.10. Whenever a symbol table is created, a corresponding `SymbolTreeNode` is added to the `SymbolTree` by default. It is put under its parent symbol table node. Because every symbol table keeps a reference to its parent, a tree structure is defined. Top level symbol tables without a parent table are added to the root of the `SymbolTree`. How parent and child tables are related is defined by implementations of `SymbolScope`, which was described in Section 3.10. It should be clear

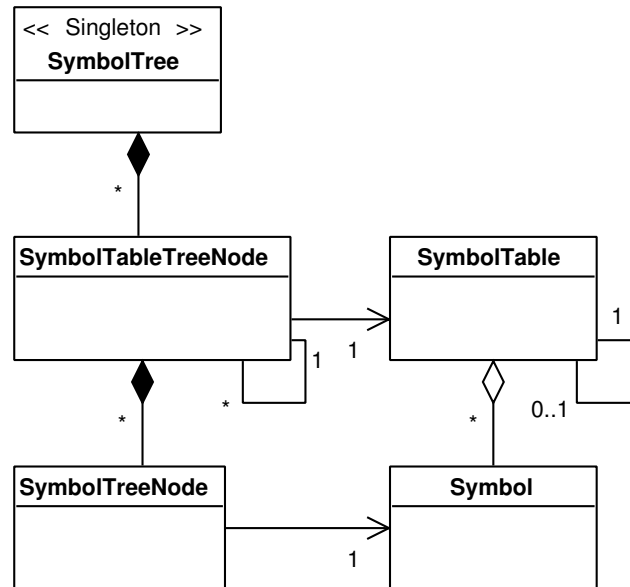


Figure 3.10: Tree related classes in Symbol Control

that typically there is only one instance of the class `SymbolTree` needed. This is the definition of a *Singleton*.

Every `SymbolTableTreeNode` that is created from a `SymbolTable` contains a `SymbolTreeNode` for each `Symbol` that is part of the symbol table. This way, symbols are added as leafes under their respective parent node. To understand the symbol tree better, a snapshot of a tree at runtime is presented in Figure 3.11. The underlying modules are a VAR and a VECM frame. Only some symbol nodes are shown for illustration. The VAR symbol table contains the variables `Y` for endogenous data, `LAGS` for the number of lags, and `RES` for the estimated residuals. Like in the previous examples, a panel for residual analysis is contained, which has itself a local symbol table to store results that are not of interest to other panels, like `P.VAL.PORT`, the p-values of a Portmanteau test. Analogous to the VAR model, a global symbol table is also used for the VECM frame. Here only the variable `RANK` for the number cointegration relations is added, together with the respective residual analysis panel.

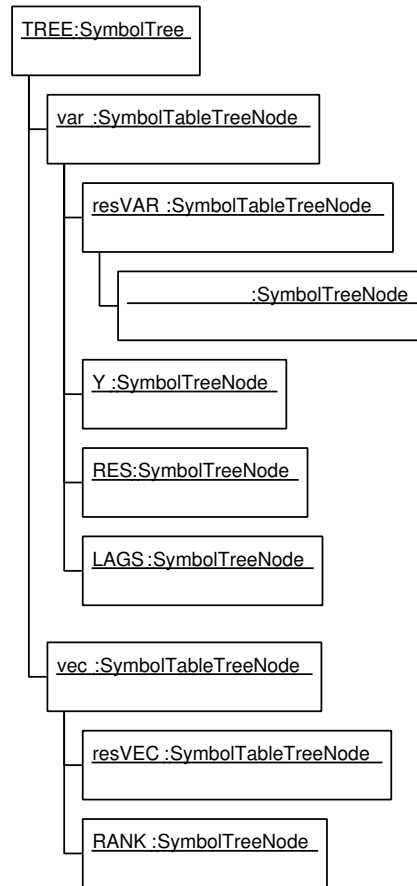


Figure 3.11: Snapshot of objects in symbol tree

VAR and VECM modules are separated from each other in the example, because many of the underlying algorithms and GUI components are different for those models. However, there are also shared classes and similar procedures. A more detailed description of how this is implemented in the reference application JMulti is given in Chapter 4.

The most important aspect of the Symbol Control system is to provide access to the described tree view at runtime via graphical user interface components. Figure 3.12 shows the related classes. There is a single instance of the class SymbolFrame, which holds the SymbolTree and instances of SymbolPanel.

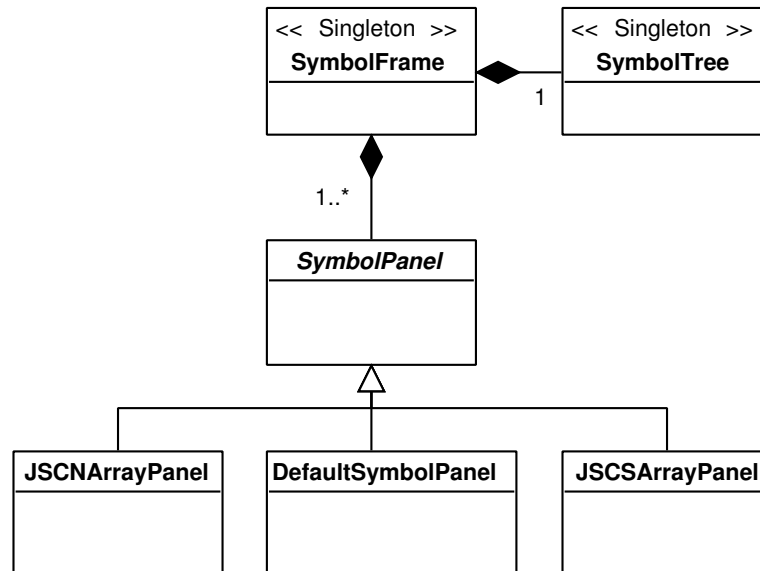


Figure 3.12: GUI related classes in Symbol Control

The symbol tree can be used to select the symbols from the hierarchy. It displays symbols with their name and an icon for the type that the referenced data has. The icons for the types are defined in the `JSCTypes` enumeration class.

The class `SymbolPanel` is an abstract class that has to be implemented by special panels to display the values of data objects belonging to a certain type. The default implementation is `DefaultSymbolPanel`, which is currently used for all types, except `NARRAY` and `SARRAY`. It merely displays the value of the data object in a text area. For the other types, special implementations of `SymbolPanel` have been created. `JSCNArray` objects are displayed in a table with adjustable precision and they can be exported to a file with several options. `JSCSArray` objects are also displayed in a table and can be exported. If a new type is added to the Type System and should be displayed, then by default the `DefaultSymbolPanel` would be used, unless a special implementation of `SymbolPanel` has been set to the `SymbolFrame` via its `addSymbolPanel` method. The abstract implementation of symbol panel allows for this flexibility, which prepares the framework for future extensions to the Data Model.

3.12.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.model.control</code>	
<code>SymbolTree</code>	Singleton ⁵ tree that contains all symbol tables as parent nodes and all <code>Symbol</code> elements as leafes.
<code>SymbolTreeNode</code>	Treenode to be used with <code>SymbolTree</code> for displaying <code>Symbol</code> objects.
<code>SymbolTableTreeNode</code>	Treenode to be used with <code>SymbolTree</code> for displaying <code>SymbolTable</code> objects.
<code>SymbolFrame</code>	A frame that holds the symbol table tree and the symbol display.
<code>SymbolPanel</code>	To be subclassed by all panels that are used in <code>SymbolFrame</code> to display a certain data type.
<code>DefaultSymbolPanel</code>	Default implementation of <code>SymbolPanel</code> to be used for all types that do not have a distinct panel.
<code>JSCNArrayPanel</code>	Panel to display and export symbols that represent <code>JSCNArray</code> data objects.
<code>JSCSArrayPanel</code>	Panel to display and export symbols that represent <code>JSCSArray</code> data objects.

Table 3.11: Elements of the Symbol Control

3.12.3 Architecture Background

Via the Symbol Control it is possible to inspect the runtime structure of all symbol tables and symbols in an efficient and convenient way. By default, every symbol table that is created, will be added to the symbol tree. Typically, symbol tables are part of the hierarchy defined by instances of `ModelFrame` and `ModelPanel` components. However, via a constructor argument a symbol table can also be created without automatically being added to the symbol tree. As already mentioned, the Symbol Management system does not need to be changed if the Type System is extended, which is a very likely scenario. The same is true for the Symbol Control. Maybe one wants to add a special `SymbolPanel` implementation for a new type, but there is always the `DefaultSymbolPanel`, which is used automatically if nothing else is specified. Thus, any new types would be displayed as well in the tree view, and user-defined types could be handled in the same way as core types that are already part of JStatCom.

3.12.4 Usage Example

The usage of the Symbol Control is demonstrated with a screenshot in Figure 3.13. It is mainly used as a readily available tool without special implications for actual code development. This is, because adding symbol tables to the symbol tree is done automatically. It can be seen that on the left side of the frame a tree view with the symbol table nodes and symbol nodes is presented. Here, the VECM frame is shown with some variables that have been set during the ongoing analysis. Gray items mark that a symbol is empty. All visible symbols are of type `NARRAY`, therefore the icons are all equal. In the upper part of the right panel, the name, type and description of the selected symbol is shown. The actual `SymbolPanel` implementation is in the lower part. It can be seen that the values of the selected symbol are presented in a table and that there are buttons for exporting the underlying data. These features would not make much sense for, say, a symbol of type `DRANGE`. Therefore for symbols of this type, the default panel would be used, as can be seen in Figure 3.14. It can also be seen, how different icons are used with `NARRAY`, `SARRAY`, and `DRANGE` symbols.

⁵A Singleton is a class that has only a single instance.

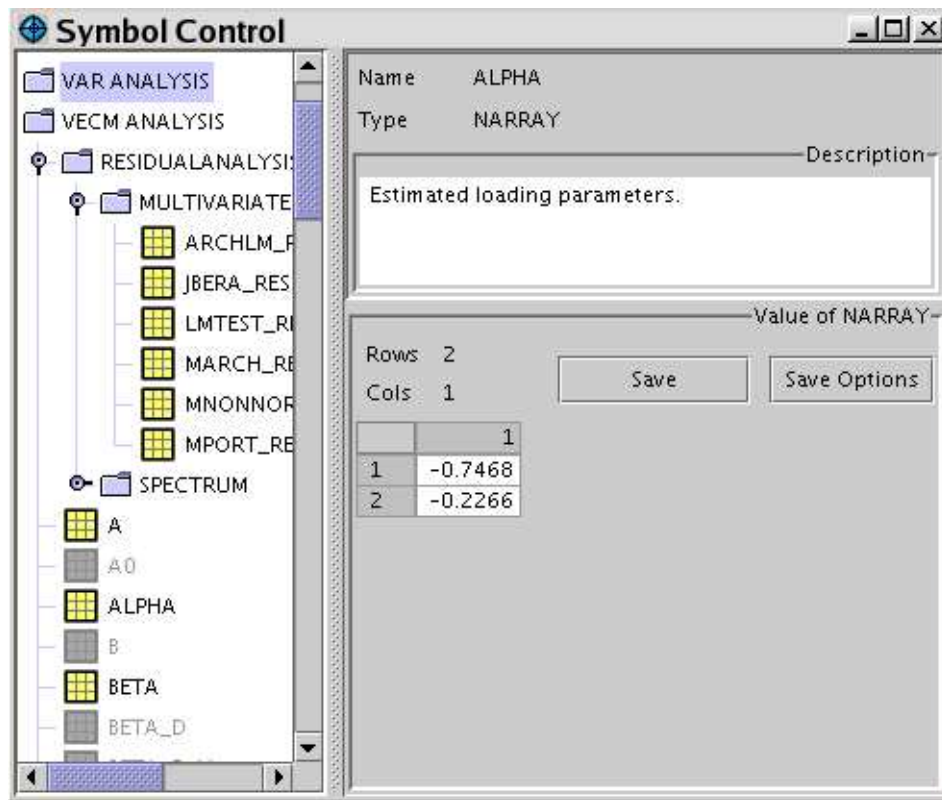


Figure 3.13: Screenshot of symbol frame with selected NARRAY

3.12.5 Related View Packets

- Parent: View Packet 2: Data Model
- Children: none

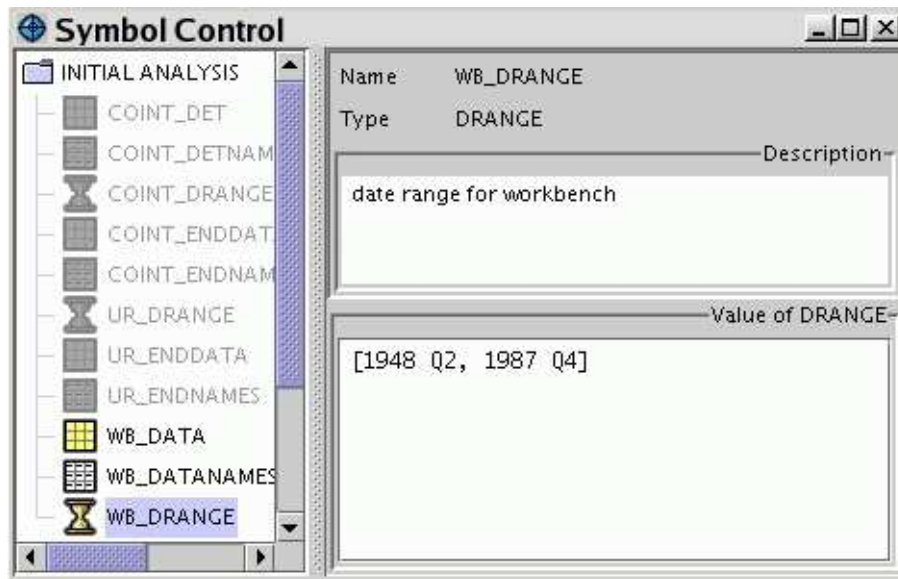


Figure 3.14: Screenshot of symbol frame with selected DRANGE

3.13 View Packet 8: Engine

3.13.1 Primary Presentation

System	Segment
Engine	Gauss
	GRTE
	Ox
	Stub
	MatLab
	PCall

Table 3.12: Primary Presentation of Engine system

This section introduces the abstract implementation of the Engine system for communicating with different execution engines. Any functionality that is specific to a certain external software is implemented in subsystems of Engine, which are shown in in Table 3.12. Typically these engines rely on external resources, which means that extra software packages or libraries must be installed.

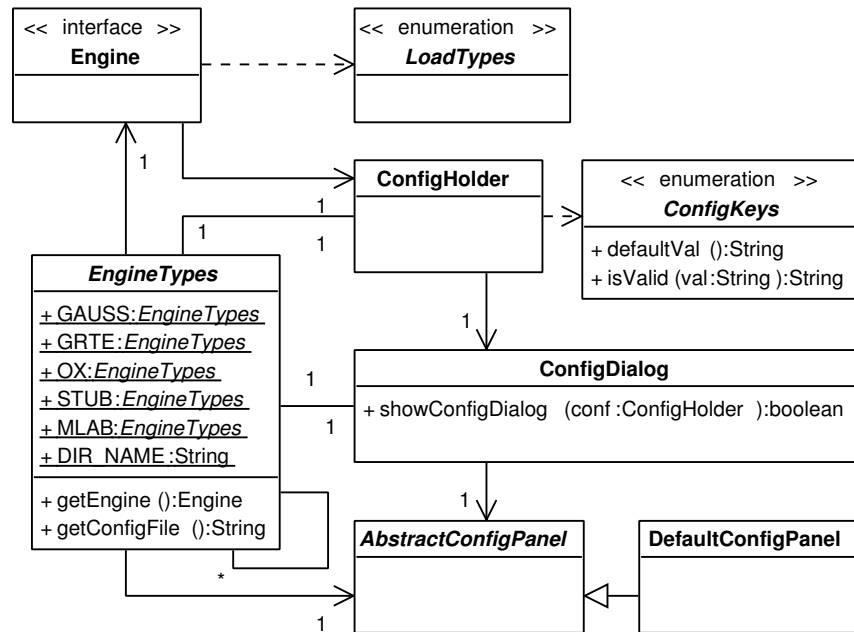


Figure 3.15: Engine classes

The main classes involved are part of the class diagram in Figure 3.15. The interface **Engine** defines the general contract that engine implementations must satisfy. Figure 3.16 shows the complete interface with all methods and all currently implemented realizations. There is one **Engine** class in each communications subsystem.

Each engine depends on several other classes which manage configuration settings and the definition of constants that are needed by a specific engine. Configurations management is a challenging task, because each engine needs different information about the computing environment. For example, the Gauss engine needs version information, the location of the executable and the location of the temporary directory. The Ox engine needs the location of the dynamic link library, whereas the MatLab engine does not need any information. The Engine system provides a mechanism to standardize the setting, storing and handling of those properties. There is one instance of **ConfigHolder** for each engine type, which retrieves the needed information from an XML file. It can also store changed set-

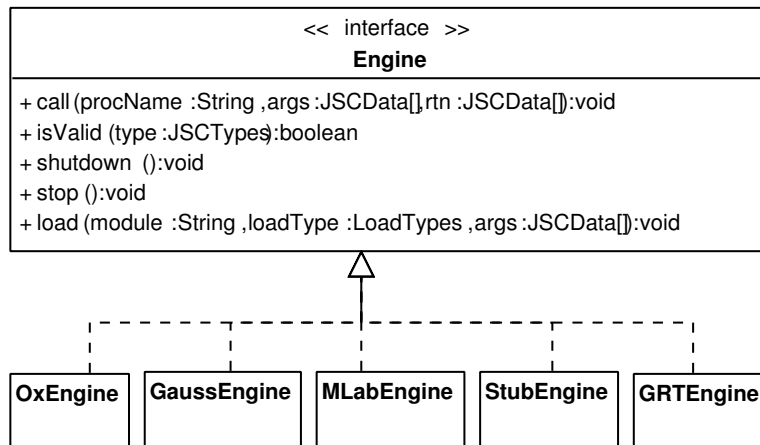


Figure 3.16: Engine inheritance

tings to that file. The format of the settings file is always the same for each engine, but it has different types of information stored in it. These types of information are defined in the abstract class `ConfigKeys`, which must be implemented for each engine subsystem to define configuration settings. Each setting is a constant with a name, a default value (implementation of `defaultVal`) and a check for correctness (implementation of `isValid`). This system gives the required flexibility to manage a different set of configuration settings for each engine. Typically this information is retrieved from the `ConfigHolder` that belongs to a certain engine type when the respective engine is initialized.

However, another part of the configurations management is that there might be information missing or wrong and must be gathered directly from the user before the engine can be executed. This is implemented via the `ConfigDialog` and the `AbstractConfigPanel`. Whenever a correctness check of a `ConfigKeys` instance fails, the mentioned dialog is presented with a panel to gather user input. This panel must be of type `AbstractConfigPanel` and can be implemented by the specific engine subsystems, because these GUI components depend on the settings that are required by the specific engine. But a default implementation is also provided with `DefaultConfigPanel`. An example of this panel is shown in Figure 3.17, because the `isValid` method of `GCG_FILENAME`, which is an instance

of `ConfigKeys` for the GRTE engine, returned an error. The user is now asked to correct the wrong file name `jgrte0.gcg`, otherwise the engine would refuse to run and throw an exception. However, other engines with correct settings would not be affected at all by this. This panel is always used when nothing else is specified. It just presents a text view of the underlying configuration file. This is of course less convenient for the user than a more customized panel, but it relieves the developer from the task of implementing such a panel in the first place.



Figure 3.17: `ConfigDialog` with a `DefaultConfigPanel`

The task of mediating between configurations management, the setting of the configurations panel for user input and initializing an engine instance, is left to the class `EngineTypes`. It must be subclassed for each engine subsystem. Typically there is only a single instance of each engine needed, which is created and referenced via the `getEngine` method of the `EngineTypes` class. The class can also be used to check configurations settings jointly, because sometimes there are certain combinations of otherwise valid single settings that are not allowed if used together. Another task of `EngineTypes` is to keep an enumeration with references to all implementations of itself in the engine subsystems, which can be seen in Figure 3.15. But the latter function is just for convenience and clarity and not at all needed for a new engine subsystem to run properly.

Resource Management

Because JStatCom is designed to handle an arbitrary number of engines at the same time, it is vital to separate the various additional resources in different locations. Those resources include the mentioned XML settings file, but also native libraries for communications interfaces and the files that contain the actual algorithms. Details differ for each engine subsystem and are described in the respective view packets. But the class `EngineTypes` provides a default for the resource directory with the variable `DIR_NAME`. All resources for an engine should go into a subdirectory starting with 'j', followed by the name of that engine which is set in the respective engine implementation, for example `jgauss`, `jgrte`, `jox`. In the same way, the configurations file is found. The default location is set in the method `EngineTypes.getConfigFile`, which returns the file name `engine_config.xml` in the mentioned subdirectory `DIR_NAME`. However, the use of these conventions depends on the implementations of engine subsystems. It cannot be enforced but it is a reasonable way to organize the needed resources.

Differences and Similarities between Engines

The idea of the whole abstract Engine system was to create a standardized interface, which is largely independent of the concrete implementations. Users should only be confronted with the `Engine` interface to call arbitrary engines, thus having to learn only a single API. However, this is a big challenge and experience has shown that it is not fully achievable, because engines differ significantly in terms of calling semantics. For example, the Ox engine allows to create objects from classes, which is not supported by the Gauss engine. Although not impossible, it would not seem reasonable to try to generalize all potential action types in a unified interface. For this reason, the engine interface provides the parameterized function `load` to address these issues. The method takes a parameter of type `LoadTypes` that defines the specific action to carry out.

Figure 3.18 gives a class diagram for an arbitrary client class that uses the Engine system. For clarity, only two concrete engine implementations are displayed. The graphic shows that clients use the abstract class `EngineTypes` and the interface `Engine` without knowing anything about the implementing classes

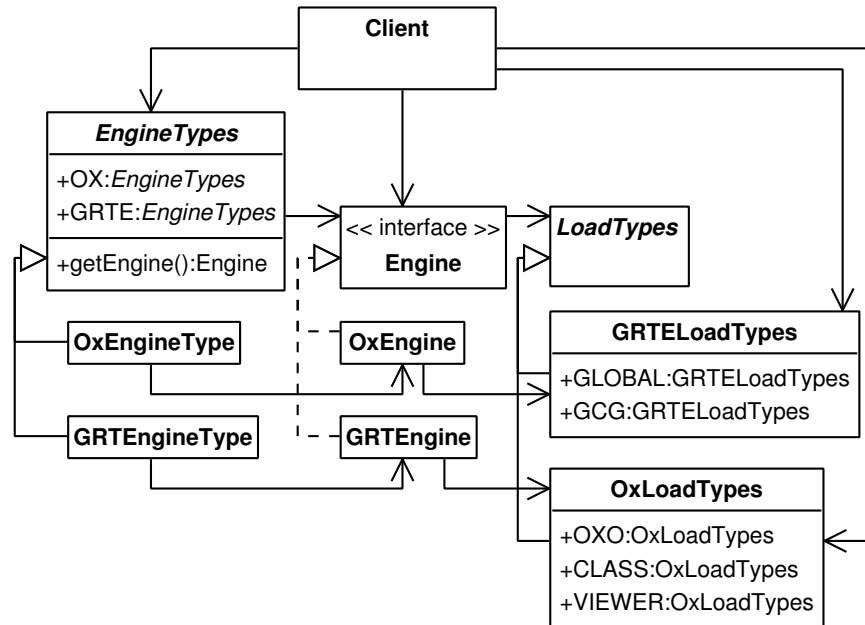


Figure 3.18: Engine client using abstract Engine and EngineTypes, but specific LoadTypes

in the background. But clients must also use the load types that are especially designed for the used engine to call the load method, thus implementation differences leak through the interface. However, this is not a severe complication, given the amount of flexibility that is achieved. Any other differences between engines are completely hidden from clients.

The solution found manages to integrate engines with very different characteristics and calling conventions. Therefore it is likely that the system will also allow to add communications interfaces to many software packages that might be used for mathematical computations. Planned extensions are the integration of R and Mathematica. This undertaking is supported by the fact that tool vendors often supply programming interfaces to control the respective software from an external application, examples are the Ox C-API, the Gauss Runtime Engine and the J/Link package for Mathematica to name just a few.

3.13.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine</code>	
<code>Engine</code>	Interface that all implemented concrete engines must satisfy. Defines the general abstract functionality of an engine.
<code>EngineTypes</code>	This class is a <i>Mediator</i> that binds the various classes and resources together that belong to an engine implementation. It also contains an enumeration of all engine types that are implemented.
<code>ConfigHolder</code>	This class is used by <code>Engine</code> instances to get all information that is necessary to run a specific engine.
<code>ConfigKeys</code>	Enumeration of constants that define different types of information needed for a certain engine to run.
<code>ConfigDialog</code>	A <code>JDialog</code> that collects configuration information about the engine from the user.
<code>AbstractConfigPanel</code>	This class should be subclassed by panels laying out an input mask to collect configuration information. It appears embedded in a <code>ConfigDialog</code> .
<code>DefaultConfigPanel</code>	Default <code>AbstractConfigPanel</code> implementation to provide a user input mask for all settings needed to run an engine. It has an editable view to the XML version of the respective <code>ConfigHolder</code> .

LoadTypes	Abstract class for defining load parameters for a certain engine. Load parameters should be implemented as enumerations that are subclasses of LoadTypes. Each load type stands for a specific operation, like loading a library or creating a class.
CPtr	An abstraction for a C pointer data type, used by the Stub engine and the MatLab engine, from the shared stubs example (Liang (1999)).
CFunc	An abstraction for a C function pointer, from the shared stubs example (Liang (1999)).
CMalloc	A CPtr to memory obtained from the C heap via a call to malloc, from the shared stubs example (Liang (1999)).
Subsystems	
Gauss	Communications package for Gauss. The software must be installed and runs in an extra process.
GRTE	Communications package for the Gauss Runtime Engine / Gauss Engine. Needs the respective vendor specific resources together with a valid license.
Ox	Communications package for Ox. The software must be installed together with packages that are needed for the modules to run.
Stub	Allows to call compiled native libraries with a basic C-interface directly from Java without the need to write a dedicated JNI wrapper.
MatLab	Allows to call compiled MatLab libraries from .m files directly from Java.

PCall	System to bundle procedure calls in a separate caller class, which is an implementation of the <i>Command</i> pattern. Can be used to run computations in an extra thread. Provides useful default functionality to execute computations in a standardized way.
-------	---

Table 3.13: Elements of Engine system

3.13.3 Context

The context of the Engine system is shown in Figure 3.19. Developers can use the system to implement calls to various engines via a standardized interface. One rather important aspect that is addressed by the PCall system is that the computations can easily be run in a new thread. This is helpful for GUI applications, because otherwise the user interface would not be reactive until the computation would have finished. Most users would consider this as unusual or erroneous behaviour. Moreover, it is one of the specific features of scientific calculations that they might be time consuming.

Certain procedures in data driven applications may fail, if the data has specific features. It is nearly impossible to guard against all possible errors with preliminary checks. Therefore a standard behaviour in case of such errors is an important aspect of the Engine system. It should give helpful feedback to the user as well as the developer, and the system should not crash but allow for correcting the error, leaving the running application in a consistent state. This is also addressed by the PCall system in conjunction with engine implementations.

How users can adjust engine settings was already described in the previous section. This is achieved via the configurations management of the Engine system.

Another desired behaviour of the Engine system is that users can stop lengthy running computations safely. However, this can not always be achieved, because it must be supported by the underlying communications system, which is typically vendor supplied. The Engine system does support this feature in general, but

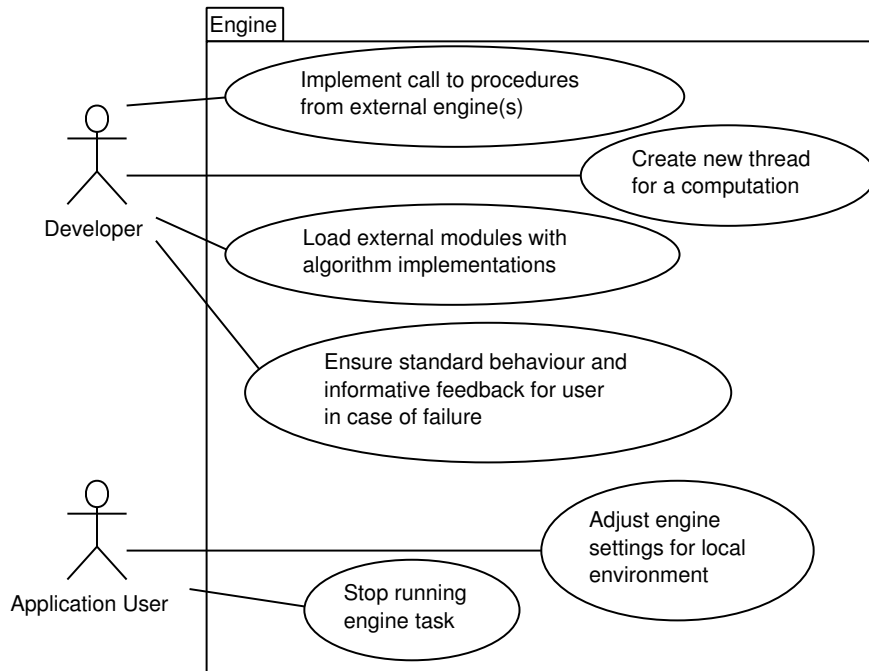


Figure 3.19: Context of the Engine system

specific engines might veto requests to stop a running procedure, simply because there is no way to stop it.

3.13.4 Architecture Background

The Engine system was designed with two main goals in mind: First, it should be easy to use for the GUI developer, who wants to implement calls to procedures that could even be written in different programming languages, and second, it should provide a framework to integrate arbitrary new engine implementations. It was therefore important to model as many similarities that engines for the given

problem domain can have in an abstract implementation, and to leave all differences to the special implementations for each engine. This has led to the current scheme with the Engine system and the specialized subsystems. The basic idea behind this separation was, that GUI developers and framework developers would benefit from it. Ideally, the GUI developer only needs to know the workings of the abstract system, thus reducing the burden of learning a different API for each system, making the resulting code easier to maintain and understand. Furthermore, the framework developer could reuse much of the general functionality, for example the configurations management, if he wants to implement a communications subsystem to a new engine. The following paragraphs describe, to which extent these goals have been met.

To achieve this, it was necessary to design a common interface `Engine` which generalizes the functionality that an external execution engine can have. This interface was subject to many modifications until it has reached the current version. The reason was that it was not general enough in the beginning to cover all variations that different engines could have. For example, in one of the earlier versions separate methods for writing data, executing methods and reading data were part of it. But this was only driven by the Gauss communications interface, which was based on the assumption that the engine runs in a separate process. When the other engines were implemented, it became clear that this separation is just a special case of the more general `call` method that is currently implemented. Another aspect of calling convenience was that the engine system must interact with the Data Model. Therefore input and return parameters are given as arrays of `JSCData`. Here the design of the Type System with the very general metadata interface has proven to be extremely useful, because only one `call` method is needed to cope with all possible procedure definitions.

Another method that was initially part of the interface was called `start`. But for certain engines this method would have no meaning, therefore it was removed from the interface. The Gauss process is now started when necessary, no client invocation is needed. However, the `shutdown` method is often helpful to clean up tasks, when the main application is exited. It can be used to remove temporary files, for example. For some engines, like the MatLab engine, `shutdown` is just empty and does nothing.

A further variation between engines is that they could accept different data types for the `call` method. For example, the Gauss and GRTE engines could potentially handle parameters with complex numbers, whereas the Ox engine does not support complex numbers. Although the complex data type does not exist yet as a `JSCData` implementation, it could be a plausible extension of the Type System in the future. Therefore the Engine System must be able to adopt to it in a consistent way. For this reason, the `Engine` interface has the `isValid` method, which should be called before any data is processed, because it checks whether the `JSCData` instances used as parameters can be handled by the implementing engine.

The discussed methods can all perfectly hide implementation details from developers. However, there are severe differences between engines that cannot be completely hidden behind an interface. There is a lot of flexibility needed to load different module types or to invoke special operations, like creating classes. This has already been described in Section 3.13.1. For this reason it was necessary to create a parametrized function `load`, which changes their behaviour according to the load parameter that is given. Therefore developers need to know the implementing engine and the semantics of the load function, because it depends on the used parameter. But typically the number of special operations is limited for each engine. An alternative would have been to find generalizations for each operation and make it part of the interface. But this would have messed up the interface with methods that are only relevant for some engines, making it harder to understand. It is argued that the current solution is the best compromise between flexibility and clarity, given the requirement that the framework should be able to potentially integrate every external engine.

Portability

Applications based on `JStatCom` are usually written in Java and are therefore portable to most operating systems. But if a certain engine is used, portability is restricted to the operating systems that are supported by that engine. Limitations stem from the fact that usually system specific dynamic link libraries are required. Those dlls might only be available to a certain operating system. `JStat-`

Com provides the source code of communications libraries that could potentially be compiled on other operating systems. This is done for the Ox engine, for example. However, some communications libraries make heavy use of operating system specific functions that are not available on other operating systems. Therefore they cannot be compiled on these systems and the source code is therefore not shipped. This is the case for the Gauss engine libraries.

3.13.5 Usage Example

Usage examples of the Engine system always rely on a certain implementation of the abstract system. Code examples are therefore given in the child view packets that describe specific subsystems. How a completely new engine communications subsystem could be set up should also become clearer by inspecting the already implemented engines, thus this is delegated to the child views as well. The reader should notice that the class structure of all engine subsystems is very similar. This is a very nice result, which shows how the abstract Engine system can be adopted to various special cases. However, the challenging part of a new engine communications scheme is often to establish the link between the external software or library and JStatCom. Although there are also patterns or best practises that can be observed, this will not be discussed in detail. In any case, it does require a solid knowledge of C/C++ programming and in the JNI.

3.13.6 Related View Packets

- Parent: View Packet 1: JStatCom
- Children:
 - View Packet 9: Gauss
 - View Packet 10: GRTE
 - View Packet 11: Ox
 - View Packet 12: Stub
 - View Packet 13: MatLab
 - View Packet 14: PCall

3.14 View Packet 9: Gauss

3.14.1 Primary Presentation

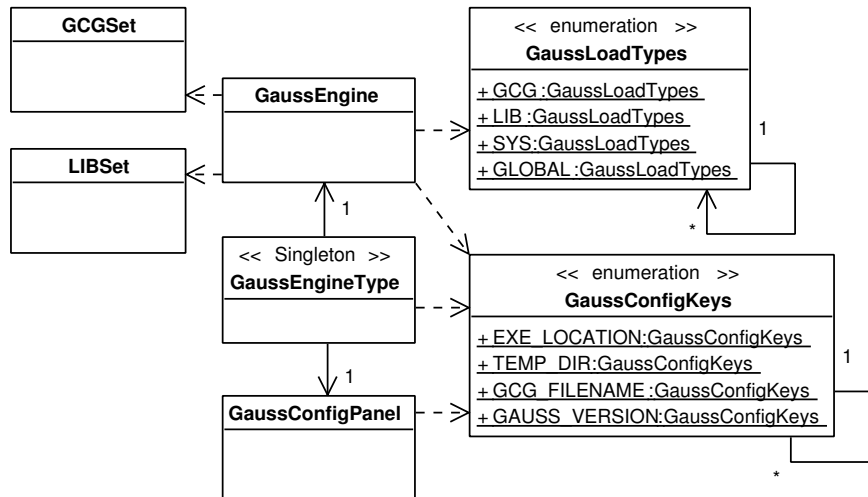


Figure 3.20: Classes of the Gauss subsystem

The Gauss subsystem is used to link between JStatCom and the Gauss for Windows software by Aptech. Supported software versions are 3.2 to 6.0. The communications libraries run only under the Windows operating system. Figure 3.20 shows the participating classes.

The class `GaussEngine` is a realization of the `Engine` interface. Via its `call` method it is possible to make calls to procedures that are defined in Gauss libraries or compiled `.gcg` files. Special Gauss modules can be loaded via the `load` function with one of the parameters defined in `GaussLoadTypes`.

The configurations management for Gauss uses the enumeration class `GaussConfigKeys` to define the types of information that are needed. Not all constants are shown in the diagram, but just the most important ones, like the location and version of the Gauss executable, the temporary directory, and the file containing the compiled Gauss code. This subsystem also provides a special version of `AbstractConfigPanel` to gather missing information from the user, which

is `GaussConfigPanel`. It is more convenient than the `DefaultConfigPanel`, especially because the settings are very likely to be changed when the system is used for the first time.

`GaussEngineType` is the class that handles initialization of the `GaussEngine` and that binds the other classes together. It also provides joint tests of the configuration settings, for example whether the Gauss version fits with the specified executable.

Two special purpose classes are `LIBSet` and `GCGSet`. Those are used by `GaussEngine` to create a set of libraries from source files and to compile Gauss sources into a single `.gcg` file.

Resources needed by the Gauss engine

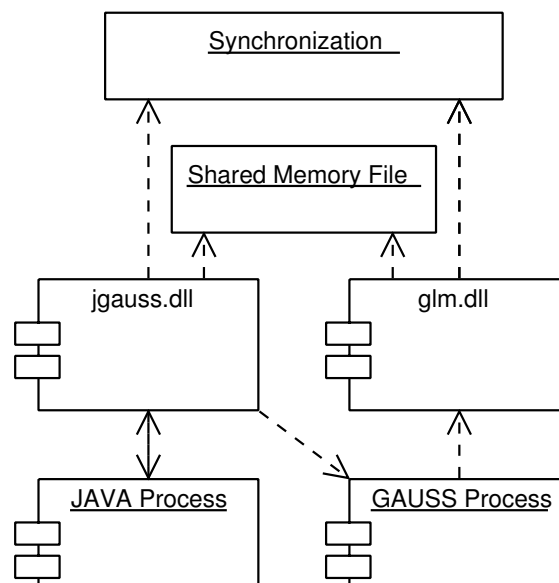


Figure 3.21: Gauss communication libraries

The basic components that enable the Gauss communication are shown in Figure 3.21. On the Java side, a process is started that is typically an application based on the `JStatCom` framework. When the Gauss engine is used, the native library

`jgauss.dll` is loaded. This library spawns the Gauss process and sets up the environment for it. The Gauss software must be installed for this to work. When Gauss is started, it must load the library `glm.dll` to enable the interprocess communication.

One problem with this approach is that there are two processes running, which have separate virtual memory. But it is necessary to transfer data between the two processes in both directions. To accomplish this efficiently, a shared memory area is used, which can be accessed via the two loaded dlls. This is a standard way of establishing interprocess communication, but it relies heavily on operating system specific functions. Furthermore, it is crucial to synchronize read and write access to the shared memory area to prevent memory corruption. Synchronization is also necessary when the Gauss process is initialized and when procedures are called. The Java side needs to wait until the procedure has finished and it needs to be notified to stop waiting.

The library `jgauss.dll` is used by the Java side, but it can also call back Java methods. This is done, when Gauss writes something to the standard output. It is then redirected to the standard output of the Java application to make it possible to read it while the application is running.

Further details of the communications system are omitted here, because it is not relevant for the developer using the framework. However, it is important to know about the resources that are shipped with JStatCom to enable the Gauss communication. Table 3.14 gives an overview of all files and how they are used.

Resource	Usage
files in working directory of JStatCom	
<code>jgauss_gcg.bat</code>	Batch file to compile source files to a single <code>.gcg</code> file, using the settings file <code>jgauss/compile.xml</code> .
<code>jgauss_lib.bat</code>	Batch file to create Gauss libraries using the settings file <code>jgauss/lib.xml</code> .

files in subdirectory `jgauss`

<code>engine_config.xml</code>	Holds all necessary information for running the Gauss subsystem. It can be edited manually or via the <code>GaussConfigPanel</code> .
--------------------------------	---

<code>compile.xml</code>	Configuration settings for compiling a <code>.gcg</code> file from the Gauss sources. Defines the name of the output file and lists all input files. Must be edited manually. Already contains default procedures for plotting which might be helpful.
--------------------------	--

<code>lib.xml</code>	Configuration settings for creating Gauss libraries from Gauss sources. Defines the name of each library and respective input files. Must be edited manually. Already contains default library for plotting which might be helpful.
----------------------	---

<code>jgauss.gcg</code>	Compiled Gauss sources. This file is loaded by Gauss on initialization.
-------------------------	---

<code>gaussXX.cfg</code>	Gauss configuration settings required by the communications libraries. <code>XX</code> is a placeholder for the Gauss version.
--------------------------	--

<code>pqgrun.cfg</code>	Graphics settings file. Required by the communications libraries.
-------------------------	---

<code>jgauss1.dll</code>	Native library that is loaded by the Java side to link to an instance of Gauss 3.2.
--------------------------	---

<code>jgauss2.dll</code>	Native library that is loaded by the Java side to link to an instance of Gauss 3.5, 3.6, 4.0, 5.0 or 6.0.
--------------------------	---

<code>glm1.dll</code>	Native library that is loaded by the Gauss side if the version is 3.2.
-----------------------	--

<code>glm2.dll</code>	Native library that is loaded by the Gauss side if the version is 3.5, 3.6, 4.0, 5.0 or 6.0.
<code>src/jgauss.src</code>	Gauss code that is loaded when the Gauss process is started. It is required to read/write string arrays from/to the shared memory area.
<code>src/extern.dec</code>	Optional Gauss sources which might be used. Defines graphics globals from <code>pgraph</code> as external. Must be used for library creation if those globals are used.
<code>src/jgrte.src</code>	Gauss source that should be compiled into the <code>.gcg</code> file if the GRTE is used with it.
<code>src/plot.src</code>	Gauss sources with some helpful plot commands. Can optionally be used.
<code>src/jmplot.dec</code>	Gauss sources with global variables that are used to adjust plot settings. Can optionally be used.
<code>src/tools.src</code>	Gauss sources with some helpful support methods. Can optionally be used.
<code>src/</code>	All other Gauss source files should be stored in that subdirectory. They can either be used via the Gauss library system or via loading a compiled <code>.gcg</code> file. It is also possible to change that subdirectory.
<code>dlib/</code>	Any native shared libraries that might be used by the Gauss procedures should be stored in that subdirectory.

Table 3.14: Resources for Gauss engine

3.14.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine.gauss</code>	
<code>GaussEngine</code>	Links to an instance of the software package Gauss. The software must be installed on the same machine and runs in an extra process.
<code>GaussEngineType</code>	Initializes the <code>GaussEngine</code> , sets the <code>GaussConfigPanel</code> and does configuration checking.
<code>GaussLoadTypes</code>	Enumeration of specific constants to define load operations for the Gauss engine.
<code>GaussConfigPanel</code>	An implementation of the <code>AbstractConfigPanel</code> to create a user input mask for configuration information related to running the Gauss program.
<code>GaussConfigKeys</code>	Enumeration of all constants that define the Gauss environment.
<code>GCGSet</code>	Stores information to compile a Gauss <code>.gcg</code> file from the source files.
<code>LIBSet</code>	Stores information to create the Gauss <code>.lcg</code> files from the source files.

Table 3.15: Elements of Gauss system

3.14.3 Architecture Background

The Gauss subsystem was the first communications system that was developed for JStatCom. It is already described in Benkwitz (2002). Since then it has been refactored to work with the new Data Model and to conform to the abstract Engine system. Because the communication to Gauss 3.2 and higher versions are very different, the required libraries have been separated in `jgauss1.dll` and `jgauss2.dll` and `glm1.dll`, `glm2.dll` respectively. This was, because the communication to newer Gauss releases runs with the batch version of Gauss, which is called `tgauss.exe` under Windows.

However, the interprocess communication poses some performance problems, which are caused by the way the processes are synchronized. It takes a fixed amount of time until the Java process is notified about the end of a procedure call. Because this problem lies in the behaviour of the Gauss batch mode, it cannot be solved by adjusting the communications libraries. As a result, computations take longer, especially if many single procedure calls are involved. This does not mean that the algorithms are executed slower, but the user experience is like that. The solution is to use a different scheme to execute Gauss procedures. A vendor supplied software is the Gauss Runtime Engine, which solves the mentioned performance problems. It is described in Section 3.15. This system also has the advantage, that the user does not need to have the Gauss software installed.

However, the Gauss system is by no means obsolete, because it serves as a platform to test and debug Gauss procedures in the first place, before the final `.gcg` file is compiled for the GRTE system. This is especially convenient, if the Gauss system is used in the so called DEBUG mode. The DEBUG flag can easily be set as a command line option when an application based on JStatCom is started. This results in a much more verbose output and it affects the behaviour of the Gauss engine. If DEBUG is true:

1. the load function together with the load parameter `GaussLoadTypes.LIB` loads a Gauss library, instead of using the compiled `.gcg` file
2. it is possible to edit the Gauss source files directly, changes take effect immediately without restarting the application if no new procedures are defined

3. all input and return parameters together with the name of the procedure are printed in the log output when the `call` method is invoked

If `DEBUG` is false:

1. the `load` function together with the load parameter `GaussLoadTypes.LIB` is ignored, nothing happens
2. changes in the source files only take effect after the `.gcg` file is compiled and the application is restarted
3. log output is only generated in case of errors

It should be noted that the libraries must first be created with the tool `jgauss_lib.bat` before they can be loaded. The tool `jgauss_gcg.bat` can be used to compile the `.gcg` file. Due to these features, the Gauss system is still in use, but mainly for development purposes. It can be used by anyone who has the Gauss software installed. When the GRTE system is applied instead, changes in the Gauss sources can only be made by the owner of the license of the respective Runtime Engine. This is somewhat restrictive, because it makes it difficult for several developers to work on the Gauss code, because changes can not easily be tested. However, if the final release is created, the GRTE is clearly a better solution, also because it runs not only under the Windows OS, but on Linux and Solaris as well. Due to the use of the `Engine` interface, it is possible to develop first for the Gauss system and switch then to the GRTE system by only changing the engine implementation that is used. The two engines have similar behaviour, except that the `load` function with the mentioned `LIB` parameter is ignored by the GRTE system. This interoperability supports the strategy to use both systems together, one for development and the other for deployment.

3.14.4 Usage Example

The following code example demonstrates a standard call to the Gauss engine. It is assumed that a library `res` has been created before with the procedure `proc(1)=nonnorm_res(res)`. The input parameter is a vector of residuals and

the function returns the test statistic and p-value of a Jarque-Bera test of non-normality. It should be noted that the load method is only executed, when the DEBUG parameter is set to true. Otherwise the method does nothing. In that case, the procedure definition must be part of a compiled .gcg file that is loaded when the Gauss engine is initialized. The default name for that file is `jgauss.gcg` in the `jgauss` subdirectory. The name can also be changed by editing the `jgauss/engine.config.xml` file.

```
// gets Engine instance
Engine gauss = EngineTypes.GAUSS.getEngine();
// initialize output
JSCData[] rtnArray = new JSCData[] { new JSCNArray("outMat")};

// generate some input data
JSCData[] inputArray = new JSCData[] { new JSCNArray("inMat",
                                                    UMatrix.rndu(100,1))};
// load the Gauss library "res", it must exist
gauss.load("res", GaussLoadTypes.LIB, null);

// call the function "nonnorm_res" defined in library "res"
gauss.call("nonnorm_res", inputArray, rtnArray);

// print the resulting array with test statistic and p-value
System.out.println(rtnArray[0].display());
```

The code could also be executed with the GRTE engine, by just replacing the first line with

```
// gets Engine instance
Engine gauss = EngineTypes.GRTE.getEngine();
...
```

The load function would always be ignored in this case. It can be seen that switching between the development engine (Gauss) and the deployment engine (GRTE) can easily be done. In the application JMulTi this is adjusted with a command line option and according to that setting either Gauss or the GRTE is used, which is as easy as changing a single line of code. However, this is not

possible between engines that execute different code, like Ox and Matlab. But the general interface makes using these rather different engines still very similar, thus hiding implementation details from the developer.

3.14.5 Related View Packets

- Parent: View Packet 8: Engine
- Children: none

3.15 View Packet 10: GRTE

3.15.1 Primary Presentation

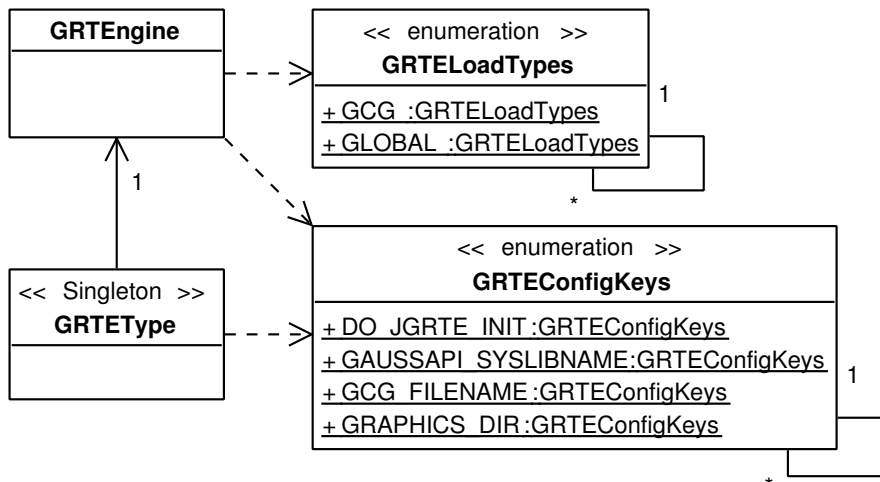


Figure 3.22: Classes of the GRTE subsystem

The GRTE subsystem is used to execute procedures written in the Gauss programming language from applications based on JStatCom. It is available for Windows, Linux and Solaris. A port to the AIX OS is theoretically possible. All participating classes are presented in Figure 3.22. The class `GRTEngine` implements the `Engine` interface, `GRTEType` subclasses `EngineTypes` and the two remaining

enumeration classes define load types and configuration keys. The GRTE engine is based on the vendor supplied Gauss Runtime Engine, which can be shipped together with the application. Thus, unlike with the Gauss communication, the user does not have to have an additional software package installed.

The GRTE engine and the Gauss engine can execute Gauss code in very similar ways. However, the details of the underlying implementations differ substantially. The GRTE runs in the same process as the Java application, therefore interprocess communication is not required. The engine manages the type conversion between the Data Model in JStatCom and the corresponding Gauss data objects.

Resources needed by the GRTE engine

To run the GRTE engine, several native libraries and resources are needed. They differ among the operating systems that the application should be run on. Tables 3.16 and 3.17 list all files that must be shipped with applications that use the GRTE via JStatCom. If an application does not use the GRTE, the listed files do not need to be shipped and can be deleted.

Resource	Usage
files in working directory of JStatCom	
mtengrt.dll	Dynamic link libraries provided by the GRTE.
opnx32.dll	Need to be located in the library search path, therefore they reside in the working directory instead of the jgrte subdirectory.
pthreadvc.dll	
files in subdirectory jgrte	
engine_config.xml	Holds all necessary information for running the GRTE subsystem. It can be edited manually or via the DefaultConfigPanel.
jgrte.gcg	Compiled Gauss code that is loaded on initialization.

<code>jgrte.src</code>	Source code that should be compiled into the file <code>jgrte.gcg</code> . It contains test methods which are used by the unit tests for the GRTE engine. Furthermore it defines the method <code>initJGRTE</code> , which is executed on initialization. This is important for graphics display on the Windows OS and for loading any dynamic link libraries that might be used by the Gauss procedures.
<code>flexlm/gauss.lic</code>	License file with which the GRTE engine runs. It is vendor provided and the compiled file <code>jgrte.gcg</code> is validated against it. If the compilation is done with different engine or with an ordinary Gauss version, the GRTE would refuse to run with that <code>.gcg</code> file.
<code>dlib/jgrte.dll</code>	Dynamic link library that exports the function <code>showLastGraphic</code> . It is loaded on initialization (see also <code>jgrte.src</code>) and required to display graphics correctly.
<code>dlib/</code>	Any native shared libraries that might be used by the Gauss procedures should be stored in that subdirectory.

cmx20.dll	Resources that are supplied by the GRTE system.
complex.fnt	
compobj.dll	
gauss.cfg	
gauss.dll	
gaussjavaapi5_0rt.dll	
import.dll	
microb.fnt	
readme.txt	
simgrma.fnt	
simplex.fnt	
symbol.gpc	
vwr.exe	
xls.dll	

Table 3.16: Resources for GRTE engine, Windows

Resource	Usage
files in working directory of JStatCom	
libmtengrt.so	Dynamic link libraries provided by the GRTE. Need to be put in the library search path.
files in subdirectory jgrte	
engine_config.xml	Holds all necessary information for running the GRTE subsystem. It can be edited manually or via the DefaultConfigPanel.
jgrte.gcg	Compiled Gauss code that is loaded on initialization. Must be compiled under Linux/Solaris.

<code>jgrte.src</code>	Source code that should be compiled into the file <code>jgrte.gcg</code> . It contains test methods which are used by the unit tests for the GRTE engine. Furthermore it defines the method <code>initJGRTE</code> , which is executed on initialization. This is important for loading any dynamic link libraries that might be used by the Gauss procedures.
<code>complex.fnt</code> <code>GaussJavaApi5.0RT.so</code> <code>libgauss.so</code> <code>libmtengrt.so</code> <code>libmteng.so</code> <code>microb.fnt</code> <code>simgrma.fnt</code> <code>simplex.fnt</code> <code>symbol.gpc</code>	Resources that are supplied by the GRTE system.
<code>FLEXlm/gauss.lic</code>	License file with which the GRTE engine runs. It is vendor provided and the compiled file <code>jgrte.gcg</code> is validated against it. If the compilation is done with different engine or with an ordinary Gauss version, the GRTE would refuse to run with that <code>.gcg</code> file.
<code>dlib/libjgrte.so</code>	Dynamic link library that exports the function <code>showLastGraphic</code> . The implementation is empty but provided for compatibility with the Windows version, which requires it to display graphics. It is loaded on initialization (see also <code>jgrte.src</code>).

dlib/	Any native shared libraries that might be used by the Gauss procedures should be stored in that subdirectory.
-------	---

Table 3.17: Resources for GRTE engine, Linux/Solaris

3.15.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine.grte</code>	
GRTEngine	Links to the Gauss Runtime Engine and conforms to the Engine interface.
GRTEType	Initializes the GRTEngine and does configuration checking.
GRTELoadTypes	Enumeration of specific constants to define load operations for the GRTE.
GRTEConfigKeys	Enumeration of all constants that define the GRTE environment.

Table 3.18: Elements of GRTE system

3.15.3 Architecture Background

The GRTE subsystem uses the vendor supplied Gauss Runtime Engine by Aptech. Because this software is available for a number of operating systems, applications based on it are not limited to a single OS anymore. However, different resources must be shipped and therefore different versions of JStatCom are available. The Java part stays always identical.

As compared to the Gauss system, the GRTE solves the mentioned performance problems and removes the restriction that Gauss must be installed before. However, there is also a drawback. Whereas running procedures can be stopped with the Gauss system, this is not possible with the GRTE anymore. The reason is that this is not supported by the programming interface for the Gauss Runtime Engine. It is hoped that this feature will be added in future releases of that software. Therefore any attempt to stop running GRTE computations results in an `UnsupportedOperationException`.

The GRTE system, as provided by Aptech, has a C-API, which can be called directly by applications written in C or C++. To call the exported functions from Java, a JNI wrapper library is required. Luckily this has been provided as well. The task of the class `GRTEngine` is therefore to serve as an *Adapter* between the Gauss Java API and the JStatCom system. The extra Java classes that are supplied by Aptech are in the archive `jars/GaussAPI5_0.jar`.

Graphics with the GRTE

The functionality of the Gauss graphics engine can also be used with the GRTE, but for the correct working some measures have to be taken. On Windows the viewer has to be invoked directly via a call to an external library. Whenever a graphics window should be displayed, the call `dllcall showLastGraphic;` must be put in the Gauss code. On Linux/Solaris this is not required, but the location of the graphics directory should be adjusted to point to a directory with write permission. In any case, the method `initJGRTE` defined in `jgrte.src` should be called on initialization. This is done by default and can be adjusted with the setting `DO_JGRTE_INIT` in the configuration file `jgrte/engine_config.xml`. It should also be mentioned that if native libraries are used from the Gauss procedures, these libraries must be loaded on GRTE initialization. This should also be done in the method `initJGRTE`.

To summarize, the following steps should be taken, when graphics are created from the Gauss code or when extra native libraries are used. They can be ignored completely if this is not planned.

1. put the line `dllcall showLastGraphic`; after a graphics window is created in the Gauss code, for example after each call to `xy` or after `endwind` if several windows are created before the graphics is shown
2. avoid overwriting the `_ptek` parameter by calling `graphset` because this would reset the setting for the graphics output file
3. adjust the file `jgrte.src` to load additional dynamic link libraries in the `initJGRTE` method, for example put the line
`dlibrary jgrte, mylib1, mylib2;`
4. compile the `.gcg` file from the Gauss sources including the file `jgrte.src`, which defines the method `initJGRTE`
5. set `DO_JGRTE_INIT` to `true` in the file `jgrte/engine_config.xml`, this is the default

Whenever a graphics window should be displayed on Windows, the resulting `.tkf` file is copied to a temporary file, which is then shown with the `vwr.exe` viewer. This viewer software is in the `jgrte` resource directory. For other operating systems, this special graphics handling is not required, because there the resulting files are in postscript and are automatically shown with the `ghostview` viewer software. However, this software must be installed on Linux or Solaris and must be found under that very name.

Portability

The Gauss Runtime Engine is available for the operating systems Windows, Linux, Sun Solaris and AIX. Therefore JStatCom could run with that engine on all of those systems. However, it is necessary to compile the `.gcg` file and the JNI wrapper dll required by the Gauss Java-API on the respective OS. The source code is not shipped with JStatCom, but is part of the Gauss Java-API distribution provided by Aptech.

3.15.4 Usage Example

Because the usage of the Gauss engine and the GRTE engine are very similar, the example in Section 3.14.4 also applies.

3.15.5 Related View Packets

- Parent: View Packet 8: Engine
- Children: none

3.16 View Packet 11: Ox

3.16.1 Primary Presentation

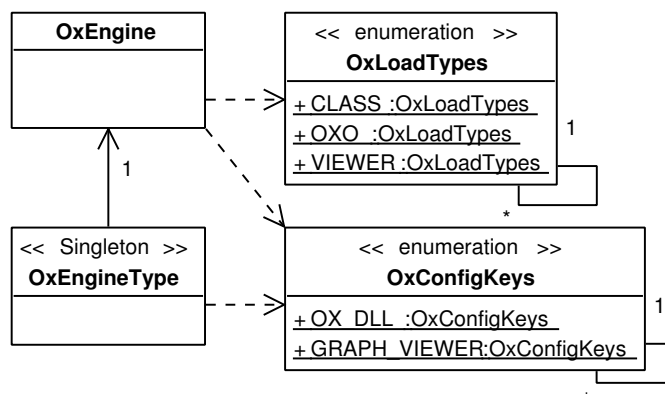


Figure 3.23: Classes of the Ox subsystem

With the Ox subsystem it is possible to instantiate Ox classes from Java and to execute member functions on them. It is required that the Ox software (Doornik and Ooms (2001)) is installed on the computer. The location of the Ox dynamic link library must be given in the settings file `jox/engine_config.xml`.

Figure 3.23 presents the classes of the Ox subsystem. The class `OxEngine` is the realization of the Engine interface, `OxEngineType` manages initialization of

the engine instance, and the two enumeration classes `OxLoadTypes` and `OxConfigKeys` define constants for the load procedure and for configuration settings.

The communication to the Ox software is set up with the help of the Ox C-API, which is especially designed for that purpose. JStatCom provides the dynamic link library `jox` to link between that C-API (Doornik (2002b)) and the Java side via JNI. This library is loaded by the Java application when the Ox engine is used for the first time. Unlike the Gauss communication, Ox does not run in a separate process, therefore interprocess communication is not required. Due to a limitation in the public Ox C-API, it is not possible to stop running Ox procedures.

Resources needed by the Ox engine

There are several non Java resources required to run the Ox subsystem. The central part of the communications interface is the dynamic link library `jox`. This library can be compiled from the source code that is shipped as well. Compilation might be necessary when the Ox engine should be run on an operating system for which no precompiled binary is provided yet, like AIX for example.

Resource	Usage
files in subdirectory <code>jox</code>	
<code>engine_config.xml</code>	Holds all necessary information for running the Ox subsystem. It can be edited manually or via the <code>DefaultConfigPanel</code> .
<code>jox.dll</code> (Windows) <code>jox.so</code> (Linux/Solaris)	The compiled dynamic link library that links Java with Ox via a JNI wrapper.
<code>test.ox</code>	Ox class that is required for running unit tests to check Ox engine functions.
<code>testextern.ox</code>	Ox procedure that is required for running unit tests to check Ox engine functions.

<code>test.oxo</code>	Compiled Ox code to run the unit tests for the Ox engine.
<code>jox_src/*</code>	Contains source code and a Makefile to compile the <code>jox</code> dynamic link library that links Java with Ox via a JNI wrapper.

Table 3.19: Resources for Ox engine, all operating systems

3.16.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine.ox</code>	
<code>OxEngine</code>	Links to the Ox software and conforms to the <code>Engine</code> interface.
<code>OxEngineType</code>	Initializes the <code>OxEngine</code> and does configuration checking.
<code>OxLoadTypes</code>	Enumeration of specific constants to define load operations for the Ox engine.
<code>OxConfigKeys</code>	Enumeration of all constants that define the Ox environment.

Table 3.20: Elements of Ox system

3.16.3 Architecture Background

The Ox implementation of the Engine system supports the following basic functions:

- loading of one or more modules (.ox or .oxo files) with class definitions and static functions
- creating an instance of a class which is then set to be the *current object*
- calling member functions of the *current object* with input arguments and return parameters of compatible types

This set of operations imposes several restrictions on the use of the Ox engine. First, it is not possible to call a static function defined in a module directly from Java. This is, because there is no API function provided that retrieves the address of a function by its name. Therefore it is necessary to define an Ox class with the needed member functions. It is then possible to create an instance of that Ox class from Java and call the provided methods. This approach can always be used to call static functions via adapter methods laid out in an Ox class.

Another limitation is that only one Ox object is referenced at a time. Whenever a new object is created, this is set to be the *current object* and all methods would be called on that one, until a new object is created. However, the Ox interface is not meant to manage the interaction of various Ox classes from Java. Instead one should think of it as an entry point to an Ox module via a single class. This class is similar to the main method of an Ox program. The used pattern is a *Facade*, because the Ox class used by the Java side can serve as an interface to a possibly complex system of Ox classes.

Furthermore, it is not possible to use all available Ox types in the methods that should be called from Java. One is restricted to numbers, arrays, strings and string arrays, which can be converted to and from Ox. Object references, function pointers or mixed type Ox arrays cannot be used. If Ox methods with these parameters must be called, the input arguments must be prepared within the wrapper class that connects Java and Ox.

The limitations discussed should not be severe in most cases. The system still provides a lot of flexibility and it should be possible to connect about any Ox module to a JStatCom GUI.

A last remark is on the use of graphics functions in Ox modules. These functions can only be linked with the Ox professional version when Ox is used via its C-API. But it is possible to install and use the GnuDraw package for Ox, which can be used to provide the graphics functions instead. The external software GnuPlot is also required. It is a powerful open-source tool, which is published under the GNU license.

Type conversion between Ox and JStatCom

Not all types that can be used within the framework have a corresponding Ox type. More generally, every engine uses a subset of all available types in JStatCom. However, a rich set of types on the Java side can make programming much clearer and tends to reduce the amount of code necessary to accomplish certain tasks. Table 3.21 shows, how types are converted to Ox values. This is always necessary, if procedures from an Ox module are called with input and return parameters. Type conversion is handled automatically by the Ox engine. There is only one limitation. It is currently not possible to produce an `OX_ARRAY` with mixed types from within Java, like `{"y", 0, 3}`. This is needed for the `Select` method of the `Ox Database` class. A workaround is to provide a wrapper class, which provides an *Adapter* to match the Java and the Ox side. This will be shown in the advanced example later on.

JStatCom Type	Represented Value	Corresponding Ox Type
JSCInt	integer value	OX_INT
JSCNumber	double value	OX_DOUBLE
JSCNArray	double array	OX_MATRIX
JSCString	string	OX_STRING

JSCSArray	string array	$n \times 1$: OX_ARRAY filled with n OX_STRING values $n \times m$: OX_ARRAY with n OX_ARRAY values, each of them filled with m OX_STRING values
JSCVoid	reference to any Java object, especially domain specific user-defined types, can be useful together with the Symbol Management to share data across components	none
JSCDate	time series date	none
JSCDRange	range marked by two time series dates	none

Table 3.21: Type conversion between JStatCom and Ox

Portability

If Ox is used as an engine, a wide range of different systems are supported. However, it is necessary to use the correct version of the dynamic link library that manages the link between the C-API and Java via the JNI. But it poses no problems to compile this library for a number of operating systems. Applications based on JStatCom that use the Ox engine can therefore be run on any platform that is supported by Ox. If GnuPlot is used for the graphics, this poses no further restrictions, because it is also available for all major operating systems.

3.16.4 Usage Example

A small code example demonstrates a typical call to the Ox engine via the Engine interface. It is assumed that the used modules exist in the OxEngine resource directory `jox`. The Ox engine also needs to know the location of the dynamic link library that contains the functions used by the Ox C-API. On Windows this library is named `oxwin.dll`, on Linux `oxl.so.3`. It must be set in the configuration file `jox/engine_config.xml` with the `OXDLL` key.

For this introductory example, a very simple Ox class is assumed. It should be defined in `jox/mymodule.ox`, relative to the JStatCom installation folder.

```
#include <oxstd.h>

class MyClass{
    decl a, x;
    MyClass(const arg);
    setX(const x);
    getX();
}
MyClass::MyClass(const arg){
    a = arg;
}
MyClass:setX(const x){
    this.x = x;
}
MyClass:getX(){
    return x;
}
```

The Java code might then be:

```
// gets Engine instance
Engine ox = EngineTypes.OX.getEngine();

// puts mymodule.ox(o) in Ox workspace, no arguments
ox.load("mymodule", OxLoadTypes.OX0, null);
```

```
// another load call, equivalent to decl x = new MyClass(3);
ox.load("MyClass", OxLoadTypes.CLASS,
       new JSCData[]{new JSCInt("arg", 3)});

// call to member function: x.setX(3.4)
ox.call("setX", new JSCData[]{new JSCNumber("x", 3.4)}, null);

// initialize result with an empty number object
JSCNumber result = new JSCNumber("result");

// call to member function: x.getX()
ox.call("getX", null, new JSCData[]{result});

// result.doubleVal() == 3.4 now
```

It can be seen that the load function is used in different ways, according to the load parameter. First the Ox module is loaded with load type OX0. Then an object is created from the class `MyClass` that is defined in the loaded module. The load type is `CLASS` then. This time the function also takes an array of `JSCData` objects as arguments for the object construction. One might argue that the load function is used for very different purposes here and that the semantics of that method changes completely. This actually contradicts the idea of an interface method, which should define a certain behaviour that is independent of the respective implementation. However, this is the point where engine differences are captured and it adds the required flexibility to the abstract Engine system. Reality bites here.

3.16.5 Related View Packets

- Parent: View Packet 8: Engine
- Children: none

3.17 View Packet 12: Stub

3.17.1 Primary Presentation

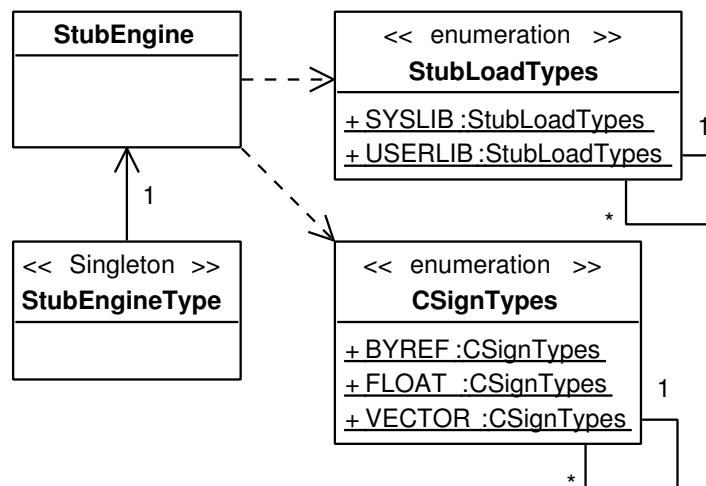


Figure 3.24: Classes of the Stub subsystem

The Stub subsystem can be used to make calls to compiled native libraries with an exported C-API directly from Java. Those libraries are typically created from C, C++ or Fortran code. Thus, it does not require a specific software product to be used as an engine, but only the respective compiler for the programming language. The major advantage of using this engine is that it is not necessary anymore to write a dedicated JNI wrapper library which links the native library with the Java side. Furthermore, it is also not required to provide any Java wrapper classes that use native methods. Therefore the Stub engine greatly simplifies calling native code from applications based on JStatCom.

Figure 3.24 shows the classes that are part of the Stub subsystem. It is not necessary to set any engine specific configuration options, therefore there is no subclass of ConfigKeys in this subsystem. The configuration file `engine_config.xml` is also not needed. The class CSigTypes is a subclass of JSCPropertyTypes that was defined in the Data Model. It can be used to set

additional properties in JSCData objects to model the signature of the C functions that are to be called.

The engine can load dynamic link libraries which export functions with a C-style signature. Table 3.22 lists the conversion rules for the JSCData parameters. It is important to understand, how data objects are converted to C-types, because this defines the signature of the native function to call. For example, if a C-function takes a `float` as an input or return parameter, then the corresponding JStatCom data object must contain this information. However, there is no JSCType that represents a `float`, only `double` by `JSCNumber` and `int` by `JSCInt`. To overcome this limitation, the Type System can be used to set special properties in JSCData objects via the interface method `setJSCProperty`. This can be used to put certain information in a data object that might be needed by engines or other components. The property does not affect the behaviour of the data object at all, but it might affect the behaviour of components using it. The Stub engine uses this system to add the needed information to the `JSCNumber` object. One could set the `CSignTypes.FLOAT` property to `true` and would therefore tell the Stub engine to convert the value to a `float` on the C side. The class `CSignTypes` is an enumeration of property types for the Stub engine. It allows to model many common C-types that are closely related to their JStatCom counterparts. However, not all types can be represented and therefore not all native functions can be called directly. But it should be relatively easy to write a wrapper in C that can be called from the Stub engine and that serves as an *Adapter* to other C-functions that use different types.

C-Type	Corresponding JStatCom Type and Properties
<code>int</code>	<code>JSCInt</code>
<code>&int</code>	<code>JSCInt</code> with <code>BYREF == true</code>
<code>float</code>	<code>JSCNumber</code> with <code>FLOAT == true</code>
<code>&float</code>	<code>JSCNumber</code> with <code>FLOAT == true</code> and <code>BYREF == true</code>
<code>double</code>	<code>JSCNumber</code>

<code>&double</code>	JSCNumber with BYREF == true
<code>double*</code>	JSCNArray with VECTOR == true
<code>float*</code>	JSCNArray with VECTOR == true and FLOAT == true
<code>double**</code>	JSCNArray
<code>float**</code>	JSCNArray with FLOAT == true
<code>char*</code>	JSCString
<code>char**</code>	JSCSArray with VECTOR == true
<code>char***</code>	JSCSArray

Table 3.22: Type conversion between C-types and Stub engine, all properties that are not mentioned must not be set or set to false

Resources needed by the Stub engine

The Stub engine requires additional resources to handle calls to the external libraries. They are listed in Table 3.23.

Resource	Usage
files in working directory of JStatCom	
<code>disp_win_i386.dll</code> (Windows)	Dynamic link library that is loaded when the Stub engine is initialized. It searches the C-function and assembles the function call.
<code>libdisp_sparc.so</code> (Solaris)	
<code>libdisp_linux_i386.so</code> (Linux)	

files in subdirectory <code>jstub</code>	
<code>stubtest.dll</code> (Windows)	Library for executing unit tests to check the operation of the Stub engine.
<code>stubtest.so</code> (Solaris)	
<code>stubtest.so</code> (Linux)	
<code>sharedstubs/*</code>	Source code with Makefile to compile the required library.

Table 3.23: Resources for Stub engine, Windows and Linux/Solaris

Portability

The Stub engine can be run on all systems for which the required dispatcher dll can be compiled. However, the code is not easily portable among different architectures, but requires rewriting the Assembler part. For JStatCom this has been done for the Linux operating system.

3.17.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine.stub</code>	
<code>StubEngine</code>	Manages calls to methods defined in native libraries with a basic C-interface.
<code>StubEngineType</code>	Initializes <code>StubEngine</code> .
<code>StubLoadTypes</code>	Enumeration of specific constants to define load operations for the Stub engine.

CSignTypes	Enumeration that defines property types for JSCData objects that can be used to model the signature of exported C functions.
------------	--

Table 3.24: Elements of Stub system

3.17.3 Architecture Background

First it should be mentioned that the so called “Shared Stubs Example” (Liang (1999)) is used to handle calls to native dlls. It consists of the Java classes CPtr, CMalloc and CFunc which are part of the general Engine system. They are also used by the MatLab subsystem. For those classes to interact with native libraries, a special JNI wrapper library is needed. This dll finds the procedure to call by its name and assembles a call from the given parameters. For this to work without knowing the explicit signature of the function to call at compile time, the actual function call is implemented directly in Assembler. Assembler is a low level language which can be used to directly address the memory of a process. It is machine dependent, because it uses the memory addressing used for a certain computer architecture. Versions of this library are currently available for Windows (i386), Linux (i386) and Solaris (Sun Sparc). Porting this library to other systems as well would require to implement the Assembler part for the respective architecture.

JStatCom uses this “Shared Stubs” system and provides a way to use it easily via the Engine interface. All parameters must be specified in an array of JSCData objects. As already mentioned, related C-types can be set via the CSignTypes properties. This is important, because the exact signature must be defined. Another peculiarity of this engine is that there is at most one return argument, because C-functions can only return a single parameter. If the return type is an array, it has to be initialized already with its correct size, because there is no way to get size information from the returned C pointer. By default, scalar input parameters are called call-by-value and array arguments call-by-reference. By using the BYREF property, also scalar arguments can be called by reference. Whenever a parame-

ter is used call-by-reference, the Stub engine updates the input parameter after the call has finished, because those parameters can be modified by the C function call. This is a common way to deal with multiple return parameters.

Using the Stub engine introduces pointer operations to the Java language. If wrong parameters are specified for a procedure or if arrays are not initialized with the correct size parameters, memory errors could occur. This would crash the Java application inevitably. Therefore careful testing is necessary when native code is called directly. However, the dangers and pitfalls of pointer operations are always inherent when C-code is called. The Stub engine only provides a way to use these procedures as well, it cannot prevent misspecification that leads to crashes. The only additional problem that it introduces is that the functions to be called are not checked at compile time, because they are assembled at runtime. Therefore it might happen that a C function is called with parameters of the wrong type or with a wrong number of parameters. This could also result in application crashes, therefore the use of unit tests to check calls made with the Stub engine is strongly encouraged.

3.17.4 Usage Example

The use of the Stub engine is demonstrated with a little example. It is possible to load system libraries that are in the library search path of the application, as well as user defined libraries. The two load types SYSLIB and USERLIB correspond to these operations. User defined libraries are searched in the `jstub` subdirectory.

In the example, a system library is loaded. The name of that library differs among operating systems, therefore the OS is determined first. Then the library is loaded, the input and output parameters are specified and the function `sin` is called. Calling this method via the Stub engine does not make much sense, because the Java language also provides trigonometric functions in its math package. However, this is just to demonstrate the workings of that system. A more likely usage scenario would be to call algorithms from the Numerical Recipes library (Press et al. (2002)).

```
// Which OS are we running on?
String osName = System.getProperty("os.name");
String libm;
if (osName.equals("SunOS") || osName.equals("Solaris"))
    libm = "libm.so";
else if (osName.equals("Linux"))
    libm = "libm.so.6";
else
    libm = "msvcrt.dll"; // Windows

// get Engine instance
Engine stub = EngineTypes.STUB.getEngine();

// load a system library, must be in library search path
stub.load(libm, StubLoadTypes.SYLIB, null);

// initialize input and output parameters
JSCNumber input = new JSCNumber("num", 2.3);
JSCNumber output = new JSCNumber("sin_num");

// call sin(2.3) via loaded system library
stub.call("sin", new JSCData[]{input},
          new JSCData[]{output});
// output.doubleVal() == 0.745705212 now
```

3.17.5 Related View Packets

- Parent: View Packet 8: Engine
- Children: none

3.18 View Packet 13: MatLab

3.18.1 Primary Presentation

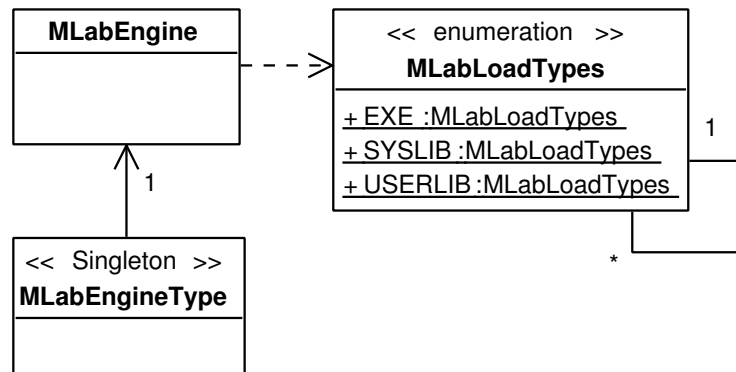


Figure 3.25: Classes of the MatLab subsystem

The MatLab subsystem is used to call dlls that have been compiled with the Matlab compiled from .m files. To do this, a license of the Matlab software together with the compiler is required. Figure 3.25 shows the classes of that system. No configuration options are required, therefore there is no subclass of ConfigKeys. The configuration file `engine_config.xml` is also not needed.

Resources needed by the MatLab engine

Because the MatLab subsystem needs to interact with native dlls, it makes use of the same library as the Stub engine. Further resources manage type conversion and are used to check the functionality of the engine.

Resource	Usage
files in working directory of JStatCom	

<code>disp_win_i386.dll</code> (Windows)	Because the Matlab engine uses the “Shared Stubs Example”, it requires the dll to manage the link between native libraries and the Java side.
<code>libdisp_sparc.so</code> (Solaris)	
<code>libdisp_linux_i386.so</code> (Linux)	
files in subdirectory <code>jmlab</code>	
<code>jmlab.dll</code>	Dynamic link library that is loaded when the MatLab engine is initialized. It manages the type conversion between JSCData objects and Matlab <code>mxArray</code> values.
<code>jmlabpath.dll</code>	Dynamic link library that is loaded when the MatLab engine is initialized. It sets the library search path, such that the dlls in the subdirectory <code>jmlab</code> are found.
<code>jmlab_src/</code>	Directory with source code to compile the required libraries <code>jmlab</code> and <code>jmlabpath</code> for other operating systems.
<code>mlabtest.dll</code>	Dynamic link library that is needed to run the unit tests for the MatLab engine.
<code>bin/</code> <code>toolbox/</code>	Resources provided by Matlab to deploy compiled dlls from <code>.m</code> files.
<code>src/</code>	<code>.m</code> files that have been used for the unittests. Also serves as an example.

Table 3.25: Resources for MatLab engine

Portability

In principle it is possible to run the MatLab subsystem on every operating system for which a Matlab compiler version exists and on which the Stub engine runs. Currently this is possible for Windows, Linux and Sun Solaris. However, because I did not have access to Matlab versions for the latter two operating systems, only the relevant source code is shipped. Compilation and testing has not yet been done.

3.18.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine.mlab</code>	
<code>MLabEngine</code>	Manages calls to methods defined in native libraries created with the MatLab compiler from <code>.m</code> files.
<code>MLabEngineType</code>	Initializes <code>MLabEngine</code> .
<code>MLabLoadTypes</code>	Enumeration of specific constants to define load operations for the MatLab engine.

Table 3.26: Elements of MatLab system

3.18.3 Architecture Background

The MatLab engine can be viewed as a special case of the Stub engine, because it is designed to call very specific C functions. This is due to the fact that the MatLab compiler generates C code from `.m` files with a distinct API. The procedures to be called take types of MatLab `mxArray` classes as arguments. Therefore the main task of the MatLab engine is to manage the type conversion between JStatCom types and the MatLab types in both directions. This is done with the help of the

jmlab dll. The mapping between types is pretty straightforward and does not require extra properties as in the Stub subsystem.

However, there is one restriction. Double indexed JSCSArray data objects are always transformed to a vector of strings when they are used as arguments for MatLab procedures. All other JStatCom data types can be represented equivalently by mxArray objects.

The use of graphics functions in compiled MatLab dlls is theoretically possible, but does not work in a satisfactory way. Tests failed to invoke graphics windows repeatedly. It might well be that this behaviour will change with future MatLab versions. A workaround is to call a compiled executable via the EXE load parameter that carries out graphics presentation. For this to work, a file with the data to print must have been created before and the name of that file must be given as an argument when the executable is called.

It is also important to ship all relevant Matlab resources in the jmlab subdirectory that are needed to run compiled libraries without a Matlab installation.

3.18.4 Usage Example

The usage of the MatLab subsystem is best demonstrated with a little example. First the file mylib.m is written in the Matlab language with a very simple OLS regression. The function ols just returns the estimators. This file could directly be invoked with the Matlab software.

```
function [b] = ols(y, x)
b = inv(x'*x)*x'*y;
// end of mylib.m
```

In a second step, the .m file is converted to C code and compiled. For this, the Matlab compiler is required, which is an extra package in the Matlab software family. It is necessary to invoke the following command from within Matlab or from the command prompt:

```
mcc -B csharedlib:mylibdll mylib
```

This creates the new dll mylibdll with an operating system specific suffix. There are also several other intermediate files generated, but only the final library

is needed. It must be copied to the subdirectory `jmlab` in the `JStatCom` working directory. A Java application could then use the `MatLab` engine to load this dll and to call the `ols` method easily:

```
// get Engine instance
Engine mlab = EngineTypes.MLAB.getEngine();

// initialize input and return parameters for regression
JSCNArray y = new JSCNArray("y", UMatrix.rndu(100,1));
JSCNArray x = new JSCNArray("x", UMatrix.ones(100,1));
JSCNArray b = new JSCNArray("b");

// load library mylibdll
mlab.load("mylibdll", MLabLoadTypes.USERLIB, null);

// call ols function
mlab.call("ols", new JSCData[] { y, x },
         new JSCData[] { b });

// print estimated coefficients
System.out.println(b.display);
```

3.18.5 Related View Packets

- Parent: View Packet 8: Engine
- Children: none

3.19 View Packet 14: PCall

3.19.1 Primary Presentation

Unlike the other subsystems of Engine, the `PCall` system does not provide another communications interface to an external software or library, but is used to manage computation requests in a standardized way. The participating classes are shown

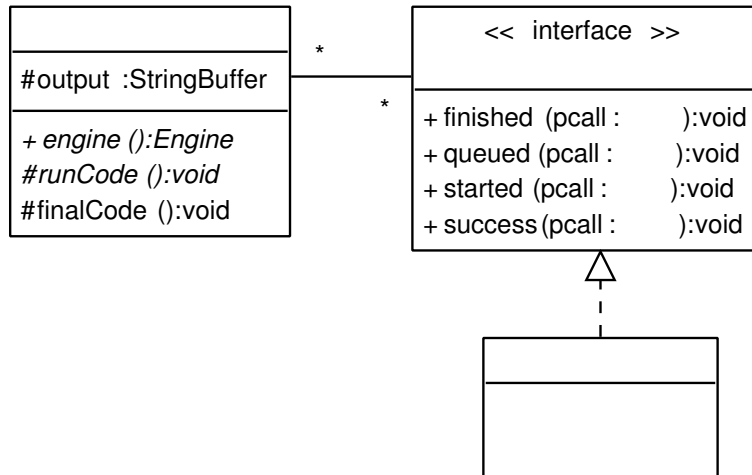


Figure 3.26: Classes of the PCall subsystem

in Figure 3.26. The core class is `PCall`, which is an abstraction for a procedure call. This abstract class should be subclassed to implement specific calls, like for example VAR estimation, the computation of a test statistic, or plotting autocorrelation functions. The implementing class encapsulates the calling logic, as well as input checking and output presentation. Furthermore, the PCall system allows to run commands in an extra background thread, such that the GUI still remains reactive, even if lengthy computations are executed.

Subclasses of `PCall` must implement the abstract methods `runCode`, and `engine`. Output formatting should typically be done in the method `finalCode`. Instances of that class can be started via `execute`. In this case they run in the background thread. However, there is not a new thread for every call, but the caller objects are queued and executed one after the other in a single worker thread. Alternatively, the `run` method can be called to execute them directly in the same thread as the calling method. The `PCall` class also provides a standardized way to handle errors that may occur in computations. In this case a message is presented to the user and error information is logged. Instances of `PCall` can be configured by setting various properties or by overwriting superclass methods in subclasses.

The other element of the PCall system is an event system that can be used to inform listeners about changes in the state of a PCall object. Classes that need to be informed must implement the interface PCallListener and register with the caller object. The methods of the interface are called, when the call is queued, started, finished and successfully finished. Typically listeners are only interested in one of those state changes, the most common are `success` and `finished`. The convenience class PCallAdapter provides empty default implementations of all interface methods. Listeners can inherit it and overwrite only those methods that are relevant for them.

3.19.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.engine</code>	
PCall	Encapsulates procedure calls to carry out computational tasks. It can optionally be executed in a single worker thread.
PCallListener	Interface to be implemented by listeners that want to be informed about changes in the status of PCall objects.
PCallAdapter	An empty convenience implementation of PCallListener.
DefaultPCallControl	Default component for displaying status information about procedure calls.

Table 3.27: Elements of PCall system

3.19.3 Architecture Background

Initially, JStatCom did not have a separate system to handle computation requests. Instead, it was integrated in the original Gauss engine implementation and could be invoked directly from instances of `GaussPanel`, the predecessor of the current `ModelPanel`. But programming experience as well as design patterns strongly encouraged to separate the distinct responsibilities. The PCall system can now be used independently of GUI components and it could also be used without applying any of the Engine subsystems. However, it provides some methods that support interaction with engines, because those systems are typically used together. The important thing is that it encapsulates algorithm calls in separate, reusable classes that can independently be invoked and automatically be tested. It is thus an example of the *Command* pattern.

The PCall system uses the concurrency pattern *Worker Thread*, which is described in Lea (2000), thus there is only one background thread which executes queued PCall instances one after the other. An alternative would have been to create a new thread for each caller object, possibly by applying the *Thread Pool* pattern. However, in an analysis the sequence of calls is often important and can cannot be changed. For example, the residual analysis for an augmented Dickey-Fuller (ADF) regression (Dickey and Fuller (1979)) can only reasonably be done after the model has been estimated. A likely scenario would be that the ADF regression is implemented in a class `ADFCall`, and the diagnostic tests for residual autocorrelation in a class `PortmanteauCall`. Both classes inherit from `PCall`.

By default, the PCall system starts executing the next thread when one call has finished, no matter whether this call was successful or erroneous. If the calls do not depend on each other, this behaviour is desirable. But in the case of the considered ADF residual analysis, the hypothetical `PortmanteauCall` should only be executed, if `ADFCall` finished successfully and the residuals are available. The PCall system provides two possibilities to cope with this requirement:

Via the add method

The instance of `PortmanteauCall` could be added to the `ADFCall` instance via its `add(PCall call)` method. Every added call is executed only, if the original

call has finished successfully. If errors occur in one of the added calls, the remaining procedures are executed anyway. The only problem with this approach is that the `PortmanteauCall` object must be created before `ADFCall` is executed, thus the residuals are not available on object construction yet. One could use a reference to the data object that will keep the residuals that are to be estimated, but if strict input parameter checks are done on object creation already, the `PortmanteauCall` object could not be created. One must be aware of these potential problems. An example code snippet could look as follows:

```
PCall job = new ADFCall(y, lags, testVersion);
job.setSymbolTable(local());
PCall jobPort = new PortmanteauCall(
    local().get(ADFTestCall.ADF_RESIDS).getJSCNArray(),
    portmanLags);
job.add(jobPort);
job.execute();
```

First the `ADFCall` object is created with endogenous data, the number of lags and the test version (constant, trend, seasonal dummies). It is assumed here that the ADF call stores the residuals in the local symbol table with the type definition `ADFTestCall.ADF_RESIDS`. The symbol table is set to the `job` object, because it needs to know where to store the results. A reference to the data object with the residuals is retrieved from there via the `get` method. On object creation of the portmanteau call this data object does not yet contain the estimated residuals, but is empty or contains previously estimated disturbances. The Portmanteau call object also takes a lag parameter. After it has been created, `jobPort` is added to `job`, which means that it is only executed if `job` finished successfully. As a last step, the ADF call is invoked, which means that it is actually queued in the background thread for execution.

Via the event system

A way that involves slightly more code but works always, is to rely on the `PCall` event system. This has the advantage that the outcome of the original call is available when the other caller objects are created. One could also invoke different

calls, depending on the state of the results. As an example, the same ADF call as before is implemented with the event system:

```
PCall job = new ADFCall(y, lags, testVersion);
job.setSymbolTable(local());
job.addPCallListener(new PCallAdapter() {
    public void success() {
        PCall jobPort = new PortmanteauCall(
            local().get(ADFTestCall.ADF_RESIDS).getJSCNArray(),
            portmanLags);
        jobPort.execute();
    }
});
job.execute();
```

Here a listener is created that acts when the estimation call finished successfully. Therefore it overrides only the method `success` of `PCallAdapter`. In this method it creates the call to the Portmanteau test. This listener is then registered with the `job` object. Only when the ADF call finishes ok, the `success` method of the listener is invoked, and the Portmanteau call is created and executed.

3.19.4 Usage Example

The following code example finishes the discussion of the PCall system. It shows the class `SPTestCall`, which implements a call to the Schmidt-Phillips unit root test (Schmidt and Phillips (1992)) via the GRTE engine. It is assumed that the Gauss procedure to be called is named `sptest`.

```
// class extends PCall
public final class SPTestCall extends PCall {
    // type definition for result data object
    public static final JSCTypeDef SP_RESULT =
        new JSCTypeDef("SP_RESULT", JSCTypes.NARRAY, "SP test Z(rho)~Z(tau)");
    private int lags = 0;
    private JSCNArray y = null;
    private JSCNArray result = null;
```

```

// constructor with arguments
public SPTestCall(JSCNArray y,int lags) {
    setName("SP Test");
    this.lags = lags;
    this.y = y;
}
// here the output formatting is done
protected void finalCode() {
    // just print test statistic
    output.append(FArg(sprintf("value of SP test statistic: %.4f\n",
        new FArg(result.doubleAt(0, 0))));
}
// here the procedure is called
protected void runCode() {
    // previous results are cleared first
    if (getSymbolTable() != null)
        getSymbolTable().get(SP_RESULT).clear();
    // initialize result object
    result = (JSCNArray) SP_RESULT.getInstance();
    // call sptest
    engine().call("sptest", new JSCData[] {y,new JSCInt("lags", lags)},
        new JSCData[] { result });
    // set result in symbol table
    if (getSymbolTable() != null)
        getSymbolTable().set(result);
}
// returns GRTE engine
public Engine engine(){
    return EngineTypes.GRTE.getEngine();
}
}

```

This class can then be invoked from GUI or other components that set the input parameters and that present the results to the user. An example client `ModelPanel` could use the following code:

```
PCall job = new SPTestCall(y, 2);
job.setSymbolTable(local());
job.execute();
```

The variable `y` is assumed to hold the series to be tested. The result is then stored in the local symbol table under `SPTestCall.SP_RESULT`. Another feature of the PCall system is that the contents of its output buffer are appended to the output holder component, if one has been set via the method `setOutHolder`. Typically this is an instance of the class `ResultField`, which is part of the Components view, see Section 3.25. Most GUI components that invoke procedures have an instance of `ResultField` to present text output of computations. They can inform the procedure call objects about this component. The call object would then print the output to that component. The modification of the above client code is just:

```
ResultField resultField = new ResultField();
... // result field is added to the GUI, omitted for clarity

PCall job = new SPTestCall(y, 2);
job.setSymbolTable(local());
job.setOutHolder(resultField);
job.execute();
```

For a description of all details of the PCall system, the API documentation should be consulted.

3.19.5 Related View Packets

- Parent: View Packet 8: Engine
- Children: none

System	Segment
Time Series	List
	Table
	Selection
	Calculator

Table 3.28: Subsystems of Time Series

3.20 View Packet 15: Time Series

3.20.1 Primary Presentation

The Time Series system provides domain-specific functionality to develop applications for time series econometrics that are based on the JStatCom framework. Figure 3.27 shows the participating classes, Table 3.28 the child systems.

Time series are represented by the class `TS`, which holds the data, the name, the type of the series, the sample range and a reference to the project. For representing dates and date ranges, the classes `TSDate` and `TSDateRange` are used. Often it is necessary to create those objects from a string representation, for example `1960 Q1`. The class to manage parsing date strings is the `TSDateParser`.

As already described in the Type System (Section 3.8), both classes also have a corresponding data type that implements the `JSCData` interface, namely `JSCDate` and `JSCDateRange`.

Any time series that is to be analysed can be added to the single instance of `TSHolder`. It is a shared repository that can be accessed by any component of the system. Typically the List subsystem is used to display all time series that are currently in the `TSHolder`.

A time series can have different types which are defined in the enumeration class `TSTypes`. For econometrics, the types *deterministic*, *endogenous* and *exogenous* are relevant. The types enumeration also defines an icon for each type. This icon can be used by components to indicate the type property of the displayed

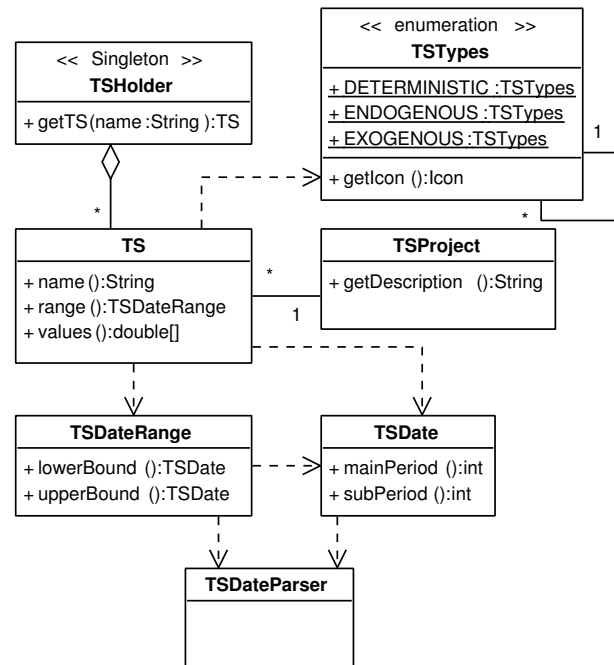


Figure 3.27: Classes of the Time Series subsystem

time series.

Finally, instances of TSPProject contain information that is common for a set of time series. Currently this is only the description for a dataset. A project also holds references to all TS objects that belong to it.

3.20.2 Element Catalog

Class Name	Responsibility
com.jstatcom.ts	
TS	Represents a time series with the observations, the date of the first observation, the name, and the project a series belongs to.

TSDate	Representation of a time series date with a main period, a subperiod, and a frequency.
TSDateRange	Range defined by two TSDate objects.
TSHolder	Holds all available time series at runtime.
TSPROJECT	Holds project information, currently this is only the description.
TSTypes	Enumeration of property types that time series can have.
<hr/> <code>com.jstatcom.parser</code> <hr/>	
TSDateParser	Parser that assembles TSDate objects from an input string.
<hr/> <code>com.jstatcom.util</code> <hr/>	
UData	Utility class with static helper methods mostly for time series analysis.
<hr/> subsystems <hr/>	
List	Subsystem that manages the representation of all available time series in a list view. This list has several components attached to it that can be used to invoke different actions on the selected time series.
Table	Subsystem that represents one or more time series in a table view. Used to display and edit data values.
Selection	Subsystem that manages the selection of time series for modelling.

Calculator	Subsystem that provides the simple language <i>TSCalc</i> to invoke various operations on a set of time series.
------------	---

Table 3.29: Elements of Time Series system

3.20.3 Context

Figure 3.28 presents the usage context of the Time Series system. Users of applications based on JStatCom access its functionality via several special purpose GUI components that help the user accomplish the tasks shown in the diagram. Those components are described in greater detail in the child subsystems. Developers use this system to represent dates and date ranges, as well as to create special time series objects.

3.20.4 Architecture Background

The Time Series system contains abstractions that are especially helpful for econometrics. During development with JStatCom it turned out that the classes `TSDate` and `TSDateRange` are used most often. For many procedures it is necessary to define a point in time or a time interval. Originally these had to be done via indices, which was extremely tedious and error prone. The abstractions for dates and date ranges make those operations much more convenient. Because they are also part of the Type System (Section 3.8), instances of these objects can easily be shared among components. It should also be noted that the date and date range classes are immutable, which means that instances of these classes can never be changed. Immutable classes have many advantages, for a complete discussion, see Bloch (2001), Item 13.

Direct access to TS objects is often not required, because the data values are typically already represented as `JSCNArray` objects, and can be retrieved via the Symbol Control (Section 3.12). The subsystem Selection manages the selection of data and stores the results in the global symbol table. Therefore, developers typically only interact with the Data Model. It should also be noted that every

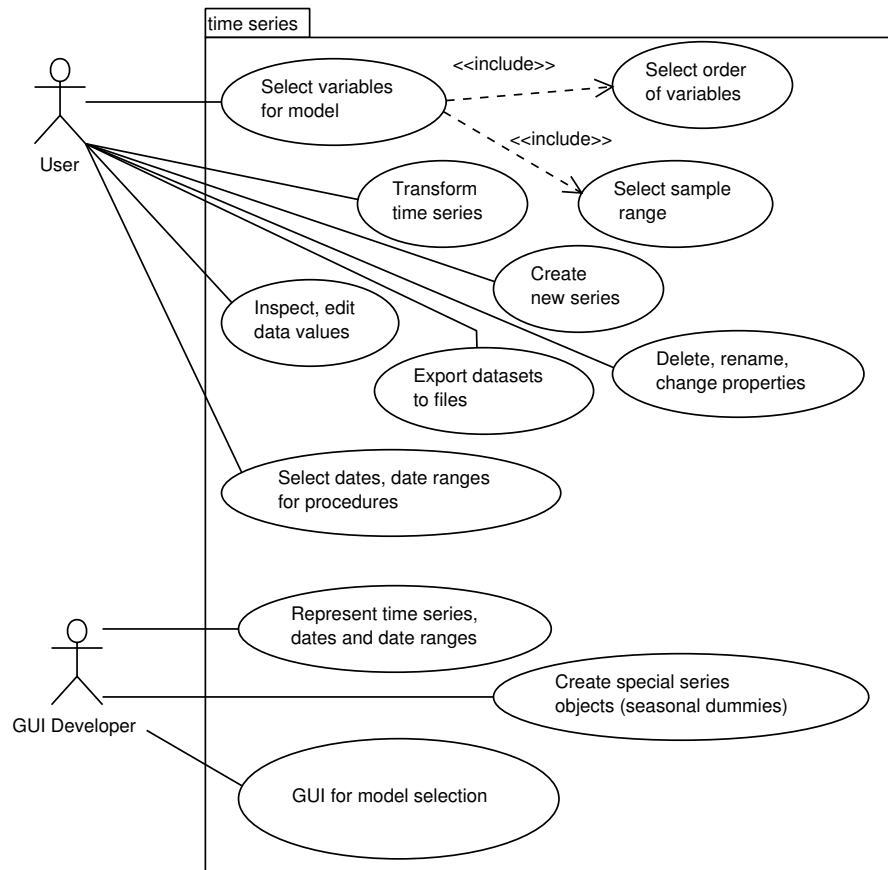


Figure 3.28: Context of the Time Series subsystem

time series is identified via its name. The naming conventions are the same as for the Type System that have been described in Section 3.8.1.

The class `TSDateParser` defines how dates can be created from a string representation. More generally, a parser is an object that recognizes the elements of a language and translates it into a meaningful result. A language is a set of strings (Metsker (2001)). The code for this parser has been automatically generated via the JavaCC tool from a grammar definition.⁶ This is a standard way of creating parser classes for Java, and the approach has been used several times during the development of JStatCom. The special package `com.jstatcom.parser` holds all generated parsers. The use of JavaCC for generation of any non trivial parsers is strongly encouraged, because it tends to give very reliable results. Problems typically result from logical errors in the grammar definition and can be detected and resolved much easier than by inspecting the source code.

The following two sections describe the “language” that is used for defining dates and date ranges. It is used by components that are part of the Selection subsystem to translate user input to date or date range objects in a reliable way.

Date Specification

Dates are objects that are defined by a main period, a frequency and a sub period. All three pieces of information must be retrieved from the input. The following formats are recognized, where D stands for the main period, S for the sub period and F for the frequency or periodicity:

- D S/F, for example 1960 1/6. This format is always possible and used as default format if no special identifier for the frequency is defined.
- D 'Q' S, for example 1960 Q 1. This format is used for quarterly data.
- D 'M' S, for example 1960 M 11. This format is used for monthly data.
- D, for example 1960. This format is used for annual data.
- D ['I', 'II'], for example 1960 II. This format is used for half-yearly data.

⁶The URL of JavaCC is javacc.dev.java.net.

- D. S, for example 1960 . 1. This format is just for input convenience and can be used when the program knows the frequency already from the context, which is often the case.

Range of Dates

The parser for date ranges is included in the method `TSDateRange.valueOf`, which takes a string argument to parse. The input string must contain two comma separated date strings, optionally enclosed by angular brackets. For the date strings, the above mentioned rules apply. Examples are:

- [1960 Q1, 1982 Q4]
- 1970 M12, 1980 M1
- 1, 100

3.20.5 Usage Example

As a little motivation, the code example demonstrates how date and date range objects can be created easily. Here two date objects are created and a range is constructed from them. Finally, the number of observations of the range is computed. There are many more helpful functions available. One should consult the API documentation for a full account.

```
// different ways to create a date representation
TSDate date0 = new TSDate(1960, 1, 4);
TSDate date1 = new TSDate.valueOf("1970 Q3");

// create new range 1960 Q1, 1970 Q3
TSDateRange range = new TSDateRange(date0, date1);
// gets number of observations for sample (T = 43)
int T = range.numOfObs();
```

3.20.6 Related View Packets

- Parent: View Packet 1: JStatCom
- Children:
 - View Packet 16: List
 - View Packet 17: Table
 - View Packet 18: Selection
 - View Packet 19: Calculator

3.21 View Packet 16: List

3.21.1 Primary Presentation

The List subsystem manages the representation of all time series that are elements of the Singleton instance of `TSHolder` in a list view. The classes of that system are shown in Figure 3.29. `TSList` is the central component of the List subsystem. It is a subclass of the standard `javax.swing.JList` class, and provides special customizations for the needs of time series analysis. Figure 3.30 shows this component during runtime. It displays the loaded time series by their names and tags their properties with icons, which are defined in the already mentioned `TSTypes` enumeration. The `TSList` class is supported by several other classes that handle mouse events and key events. A special mouse event triggers the popup menu. There are default listeners installed that are shared by each instance of `TSList`. This is to ensure a consistent behaviour of the list view, because typically it appears several times in an application. However, the default listeners can be replaced as well.

The class `DefaultTSListPopup` is also shown in action in Figure 3.30. It is a menu that consists of items that invoke actions, as well as submenus that contain action items themselves. The actions typically operate on the selected time series in the underlying `TSList`. All actions are enumerated in the class `TSActionTypes`. Diagram 3.29 only displays three of them for clarity. It is possible to define own action types and to add them to the `DefaultTSListPopup`.

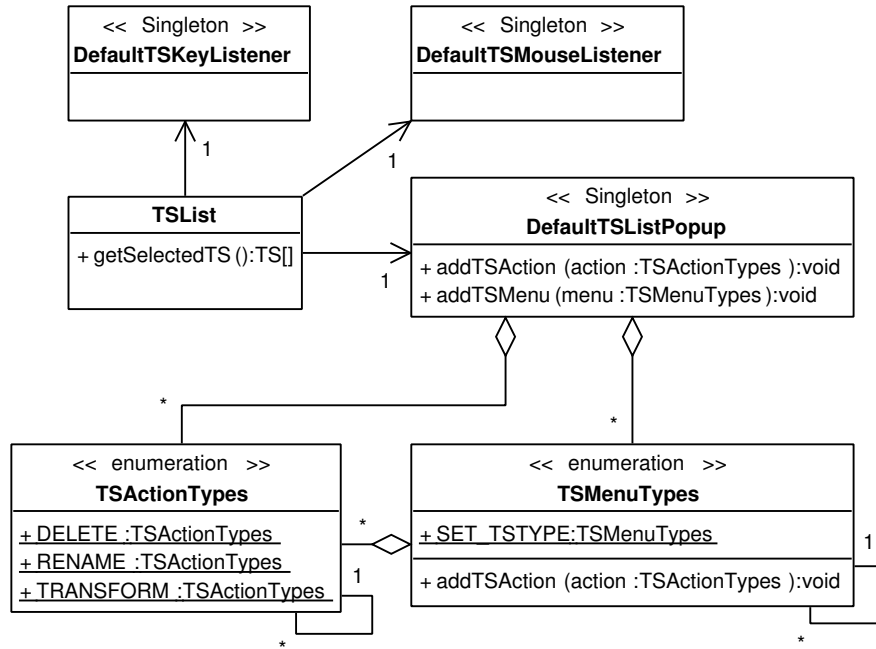


Figure 3.29: Classes of the List subsystem

This extends the default behaviour for lists used in a specific application. In Figure 3.30 an extension to the default popup menu is the *Plot* menu item. It is not part of the default actions, because its implementation in the shown example depends on the availability of the GRTE engine, which cannot always be assumed for all applications based on JStatCom. This is, because if developers want to program with the GRTE to integrate their own Gauss code, they need to own a license of that software. Furthermore, the GRTE might not be available for the platform to run the application on, for example if the target platform would be a Mac OS.

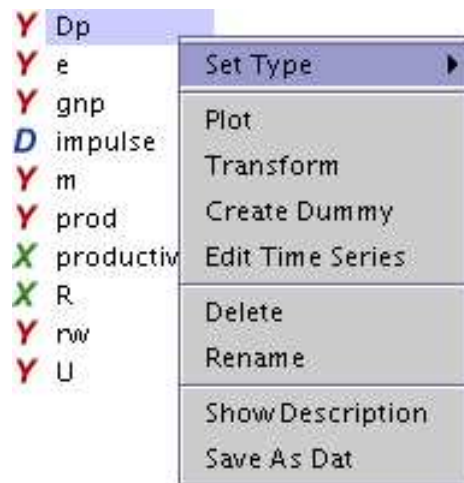


Figure 3.30: Screenshot of a TSList component with TSListPopup showing

In the same way as actions can be added, it is possible to define and add new submenus to the default popup and to fill them with specific action items. It should be mentioned that the List subsystem is usually not used directly for building applications for time series analysis. Instead, the TSSel class of the Selection system is applied, which uses the List subsystem for displaying the available time series. However, it is still important to understand it, if the default behaviour of the List system should be changed. It is of course also relevant for maintainers of the framework.

3.21.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.ts</code>	
<code>TSList</code>	Provides a list view of all available time series.
<code>DefaultTSMouseListener</code>	Default mouse listener that is shared among all instances of <code>TSList</code> . Currently this listener does nothing, because no special mouse behaviour is attached to a time series list (only popup menu triggering). The class might be extended in the future to allow for a configuration of the standard behaviour. The default listener can also be replaced via <code>TSList.setTSMouseListener</code> for single instances of <code>TSList</code> .
<code>DefaultTSKeyListener</code>	Default key event listener that is shared among all instances of <code>TSList</code> . Currently it listens to <code>KeyEvent.VK_DELETE</code> events and deletes all selected series. The class might be extended in the future to allow for a configuration of the standard behaviour. The default listener can be also replaced via <code>TSList.setTSKeyListener</code> for single instances of <code>TSList</code> .
<code>DefaultTSListPopup</code>	Configurable default popup menu that is triggered when the right mouse button is clicked over a <code>TSList</code> . It is shared among all instances of <code>TSList</code> . It is possible to add extra menus and actions that will appear in all list instances that use it. The default popup can also be replaced via <code>TSList.setTSListPopup</code> for single instances of <code>TSList</code> .

<code>TActionTypes</code>	Enumeration class with action instances that define items in the shared <code>DefaultTListPopup</code> . This class can be subclassed to define other actions that should appear in the list popup menu by default. Those actions should then be added at program start to the default popup via <code>DefaultTListPopup.addAction</code> .
<code>TMenuTypes</code>	Enumeration class with <code>JMenu</code> instances that define menus in the shared <code>DefaultTListPopup</code> . This class can be subclassed to define other menus that should appear in the list popup menu by default. Typically a menu contains instances of <code>TActionTypes</code> . Those menus should then be added at program start to the default popup via <code>DefaultTListPopup.addTMenu</code> .
<code>TSDummyDialog</code>	A dialog for creating dummy variables. It also invokes the desired action and adds the created time series to the underlying <code>TList</code> .
<code>TSEditDialog</code>	This component provides editing capabilities for selected time series in a <code>TList</code> .
<code>TSTransDialog</code>	Dialog for specification of basic time series transformations. It also invokes the desired transformation and adds the created time series to the underlying <code>TList</code> .
<code>TSDescDialog</code>	Dialog for entering a description for a project, before it is saved to a <code>.dat</code> file.

<code>TSListAccessor</code>	Interface used by <code>TSList</code> to enable/disable components accessing it. Only for internal use, currently implemented by <code>TActionTypes</code> and <code>TMenuTypes</code> to change its enabled state according to the selection state of the list.
-----------------------------	--

Table 3.30: Elements of List system

3.21.3 Architecture Background

The List system plays a central role for providing GUI components that facilitate time series analysis. It is used for selecting variables and for invoking various actions on the selected series. Whenever the user needs to choose from a set of series, typically the List system is involved.

When the system was designed, it became clear that there are several potential extension points where developers might want to add or remove features. Therefore some effort has been invested to provide a relatively simple way to change the default behaviour of the list, as well as to change the behaviour for single list instances to deviate from the default. The solution was to use configurable and shared default instances of the relevant listeners and involved components. This is also more efficient than to provide a new instance of those classes for every list that is created. So far, only the `DefaultTSListPopup` class can be configured by adding and removing actions or menus. However, `DefaultTSMouseListener` and `DefaultTSKeyListener` seem to be reasonable choices that do not necessarily need to be changed. Those classes might also be extended in the future to allow for changing the default behaviour.

In any case, it is also possible to not use the default classes and to replace them with own implementations for certain instances of `TSList`. For example, it is sometimes useful not to have a popup menu. This can easily be achieved by using the `TSList.setTSPopup` method with a `null` argument. All other list instances would not be affected by this customization, because the default behaviour would remain the same.

3.21.4 Usage Example

The following example demonstrates the definition of a new `TActionTypes` instance, which is then added to the `DefaultTSListPopup`, thus changing the default behaviour of the list view. This customization should be done at application start. The example is taken from the `JMulti` application, which is described in greater detail in Chapter 4.

First, the class `TActionTypes` is subclassed to implement the plot functionality. The method `doAction` is invoked when the popup menu is clicked with the underlying `TSList` instance as argument. This way, the method can get access to the selected time series, making it easy to invoke the plot action. The details of the plot implementation are omitted here. One might ask, why this action is not part of the default implementation. The answer is, because it relies on an external engine, which is not visible from the presented code snippet. However, it might become part of the default implementation, if a powerful enough Java library is found to invoke the plot independently of any external resources. The most promising candidate for this is the *jfreechart* library.

```
// implementation of a new action that invokes a plot
public final class TSPLOTAction extends TActionTypes {
    // Singleton instance
    public static final TSPLOTAction TSPLLOT = new TSPLOTAction("Plot");
    // constructor
    private TSPLOTAction(String name) {
        super(name);
    }
    public void doAction(TSList tsList) {
        TS[] sel = tsList.getSelectedTS();
        // invoke plotting of all selected series
        ...
        // details omitted for clarity
    }
}
```

The new plot action `TSPLOTAction.TSPLLOT` can then be added to the shared instance of `DefaultTSListPopup` at program start via:

```
DefaultTSListPopup.getSharedInstance().addTSActionAt(
    TSPlotAction.TSPLOT,
    2);
```

This way, all list views would automatically get the *Plot* item at index 2 in the popup menu of the *TSList*.

3.21.5 Related View Packets

- Parent: View Packet 15: Time Series
- Children: none

3.22 View Packet 17: Table

3.22.1 Primary Presentation

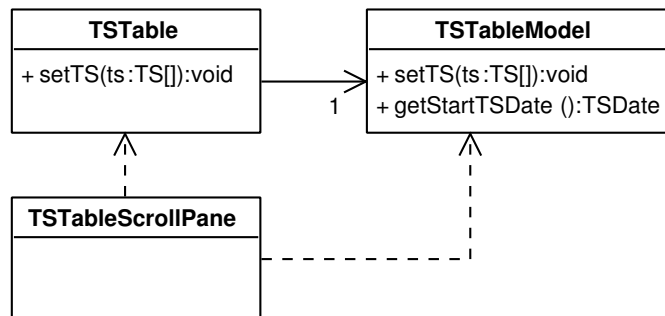


Figure 3.31: Classes of the Table subsystem

The Table system is used to display a set of time series in a spreadsheet that can optionally be edited. Figure 3.31 presents the participating elements. It consists of the class *TSTable* and a special table model *TSTableModel*. An array of time series objects can easily be set for display via the method `TSTable.setTS`. The special features of this table are that the time series are automatically merged to a range that covers all observations of all displayed series. This way, one can

Time / Index	e	prod	rw	U	
1994 Q1	57	946.7800	408.5785	458.8024	10.9700
1994 Q2	58	947.6283	409.6767	459.0564	10.6300
1994 Q3	59	948.6221	410.3858	459.1578	10.1000
1994 Q4	60	949.3992	410.5395	459.7037	9.6700
1995 Q1	61	949.9481	410.4453	459.7037	9.5300
1995 Q2	62	949.7945	410.6256	460.0258	9.4700
1995 Q3	63	949.9534	410.8673	460.2853	9.5000
1995 Q4	64	950.2502	411.1090	460.5448	9.2700
1996 Q1	65	950.5380	410.3507	460.8043	9.5000
1996 Q2	66	950.7871	410.5924	461.0638	9.4300
1996 Q3	67	950.8695	412.1160	464.6888	9.7000
1996 Q4	68	950.9281	412.9994	465.0717	9.9000
1997 Q1	69	951.8457	412.9551	464.2851	9.4300
1997 Q2	70	952.6005	412.8241	464.0344	9.3000
1997 Q3	71	953.5976	413.0489	463.4535	8.8700
1997 Q4	72	954.1434	413.6110	465.0717	8.7700
1998 Q1	73	954.5426	413.6048	466.0889	8.6000
1998 Q2	74	955.2631	412.9684	466.6171	8.3300

Figure 3.32: Screenshot of a TSTable component

display and edit time series with different sample ranges jointly. Furthermore, the precision can be adjusted, and a default popup menu over the table allows to change the display on the fly.

TSTableScrollPane is a customized subclass of the standard JScrollPane that manages the display of a row and column header for the displayed time series. A screenshot of the Table subsystem components is given in Figure 3.32. Here it is embedded in the time series editor, which is a component from the Selection subsystem. All selected time series are shown and can be edited. The editor also provides a search option, which uses the `findRowIndexOf` method of the TSTableModel.

3.22.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.ts</code>	
<code>TSTable</code>	A configurable table view of one or more time series with editing capabilities. It uses the <code>TSTableModel</code> to represent the underlying data.
<code>TSTableModel</code>	A table model to be used with the <code>TSTable</code> component for representing merged time series data. The model takes an array of time series with possibly different time ranges, but equal sub periodicities.
<code>TSTableScrollPane</code>	A special <code>JScrollPane</code> to work with <code>TSTable</code> components as viewports that provides a row header with time and index, and a column header with the names of the displayed series.

Table 3.31: Elements of Table system

3.22.3 Architecture Background

The Table subsystem handles the specific needs for displaying and editing time series. It follows the standard Model-View-Controller pattern, which separates representation, display, and control of the data. The class `TSTable` manages display and control, whereas the class `TSTableModel` stores the underlying data and informs the presentation component about updates. This scheme follows standard practise for designing components that are based on the Java Swing component framework.

3.22.4 Related View Packets

- Parent: View Packet 15: Time Series
- Children: none

3.23 View Packet 18: Selection

3.23.1 Primary Presentation

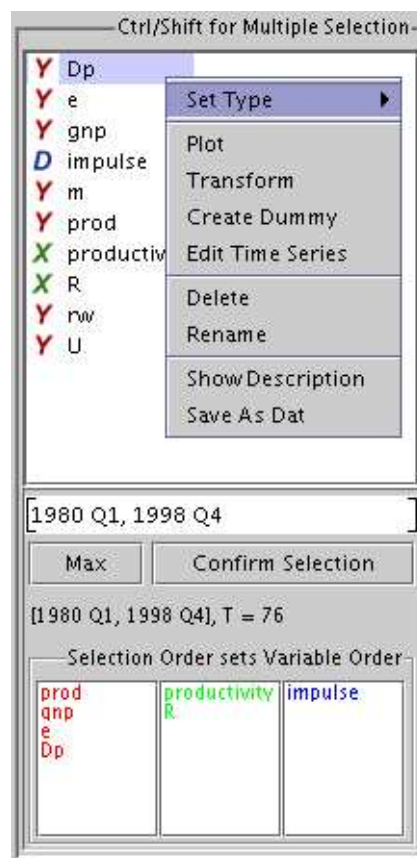


Figure 3.33: Screenshot of a TSSel component

The Selection subsystem contains components that are especially useful to gather user input for time series analysis procedures. It consists of the time series

selector, the date selector, and the date range selector. Figure 3.33 shows the first of these three components, which is the most complex one and actually contains a `TSDateRangeSelector` instance. The selection classes are all components according to the JavaBeans specification (Sun Microsystems (1997)), making it easy to plug them together with other components with the help of a graphical programming tool. The latter should be part of the employed IDE. This means that instances of those classes can be configured with various different options which affect the behaviour of the respective component at runtime.

Time Series Selector

Once a dataset has been read in, the single variables can be accessed via instances of the `TSSel` class. All time series appear in a `TSList`, which is part of the selector. The behaviour of the `TSList` has been described in Section 3.30. It allows to select variables and invoke actions on them via the right click popup menu. This is a typical example of object composition, because the time series selector *has* a `TSList`, instead of subclassing it to get access to the functionality. Especially for GUI design the advantages of object composition become very obvious, but they translate equally to the design of non graphical classes.

The time series selector is used to select the set of variables, the order of the variables, and the time range for the various econometric methods and models. The selection mechanism can be adjusted to its context, which means, for example that for unit root tests only one variable can be selected, whereas for VAR modelling the number of variables is not restricted. The respective setting of the `TSSel` bean is `setOneEndogenousOnly`, which is `false` by default, but can be set to `true` to allow only selection of a single endogenous variable.

Sometimes the ordering of the variables is important for a model or analysis. The selector uses the order in which the variables have been clicked on. For multiple selection it is necessary to hold either the `Shift` or the `Ctrl` button pressed, while the mouse is clicked over a series. The selected variables are displayed in the correct order in the control area of the selector after the selection has been confirmed.

The time range of the selection can be adjusted by editing the respective text field, which is an instance of `TSDateRangeSelector`. The valid range is the largest common range of all selected time series, where NaN missing values have been automatically truncated from the beginning and the end of each series. The smallest legal range must contain two observations. Once a range selection is made, this selection is kept as long as it is valid. The maximum possible range can easily be set via a button. This mechanism enables the user to keep an edited time range as long as possible, but allows a quick switch back to the maximum range as well.

Date Selector

The date selection component is used, whenever the user should select a single point in time, for example to specify the impulse time of a dummy variable. It uses the date format that has been described earlier in Section 3.20.4. The input is validated against an enclosing range if one has been set. Descriptive error messages help the user to correct misspecifications.

Date Range Selector

The date range selection component is very similar to the date selector, but it takes two comma separated date strings as input and assembles a date range object from it. This can also be validated against an enclosing range. Error messages direct the user in case of wrongly specified range strings.

3.23.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.ts</code>	
TSSel	Configurable selection tool for time series. It manages the selection of endogenous, exogenous and deterministic variables and builds the corresponding data objects to be reused by other components via the symbol table. Selection is only allowed for series with the same periodicity and at least 2 observations time overlap.
TSDateSelector	Component for display and selection of a single TSDate. It takes an input string and validates it against a parser and a defined TSDateRange.
TSDateRangeSelector	Component for display and selection of a TSDateRange. It interprets an input string and validates it against a parser and a defined range.

Table 3.32: Elements of Selection system

3.23.3 Architecture Background

All three classes of the Selection subsystem are meant to facilitate creating GUIs for time series procedures. They are high level components that make use of the services provided by other subsystems. Especially the TSSel class can only work in conjunction with the Data Model, because it creates a number of JSCData objects, which are then automatically stored in the global or upper symbol table. This way it is possible that other components can easily access the selection made by this component via the Symbol Management system.

3.23.4 Usage Example

The time series selector is an indispensable part of GUIs for time series analysis, therefore its use is demonstrated with a small code example. One should notice that this component does not necessarily need to be configured manually, but that visual programming tools can be used instead. For a comprehensive description of all possible configuration options, one should refer to the API documentation.

```
// creates instance
TSSel sel = new TSSel();

// sets the names of the symbols under which
// the variables names are stored in the global symbol table
sel.setEndogenousStringsName("Y_names");
sel.setExogenousStringsName("X_names");
sel.setDeterministicStringsName("D_names");

// sets the names of the symbols under which
// the data objects are stored in the global symbol table
sel.setEndogenousDataName("Y");
sel.setExogenousDataName("X");
sel.setDeterministicDataName("D");

// sets the name of the symbol under which
// the selected sample range is stored in the global symbol table
sel.setDateRangeName("SAMPLE_RANGE");
```

The names for the various data objects should be chosen according to the modelling context. They will appear under that name in the Symbol Control system. For more complex GUIs it is strongly suggested to use a separate class for defining all global variables as instances of JSCTypeDef. One should then reference those type definitions for the configuration of the TSSel component. Otherwise one can easily introduce errors by just changing a variable name in one place, but forgetting to change it in another place that refers to it as well.

3.23.5 Related View Packets

- Parent: View Packet 15: Time Series
- Children: none

3.24 View Packet 19: Calculator

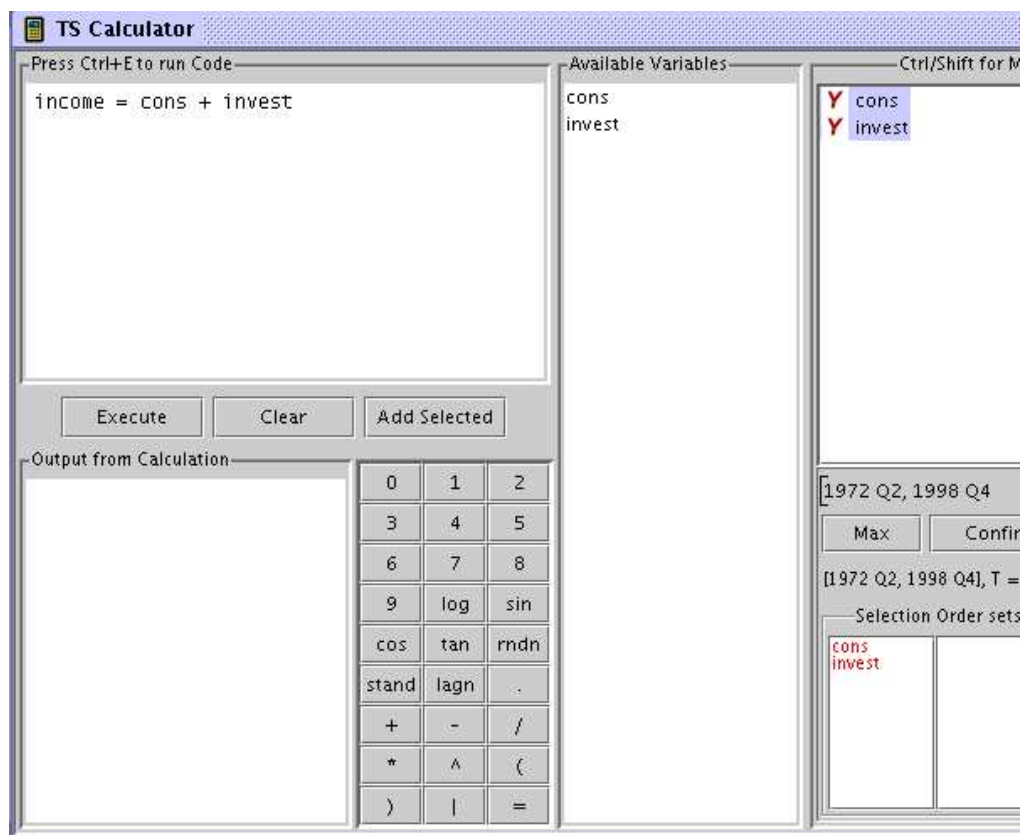


Figure 3.34: Screenshot of Time Series Calculator

3.24.1 Primary Presentation

The time series calculator is a very flexible tool to create new variables by combining existing time series with arithmetic operations and functions. It provides

the mini language *TSCalc* that operates with one dimensional arrays and scalars. The operation is as follows:

1. First one has to select the variables to be combined in the time series selector.
2. The names appear in the list `Available Variables` and are put into the variable space of the calculator.
3. One can write one or more commands to the command area and execute them with `Execute` or `Ctrl+E`. Newly created variables appear in the list of available variables.
4. Finally the selected variables in `Available Variables` can be added to the workspace with `Add Selected`.

The syntax of *TSCalc* is very easy, only a few simple rules apply:

- New variables can be defined with with `newvariable = some expression;`.
- Several commands can be executed at once by separating them with `' ; '`.
- Allowed operators are `' + '`, `' - '`, `' * '`, `' / '`, as well as `' | '` to append one vector to another.
- The content of a variable can be printed out by just writing the variable name to the command line.
- The conventions for variable names hold as described in Section 3.8.1.
- All array operations are applied element wise.
- *TSCalc* understands exponential notation, for example, `1.234e-3` is parsed to valid number.

By double clicking on a variable in the list, the name appears in the command window and can be combined with commands from the calculator. Apart from basic arithmetic operations, *TSCalc* provides a range of other functions, like `min`,

max, lagn, log, stdc, meanc, rndn, ones, trend. For a complete list of possible operators and functions, one should consult the help system. In case there are syntax errors, a descriptive message is printed to the output window. If the selection in the time series selector is changed, the workspace of the calculator is overwritten. Variables that have not been added to the workspace are lost.

3.24.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.ts</code>	
TSCalculatorPanel	A GUI component for the CalcEngine parser. It makes it possible to create new time series by combining existing ones with arithmetic operations and functions.
TSCalculator	A component that combines the TSCalculatorPanel with a TSSel component to select the time series available for transformation.
TSCalcFrame	Frame containing the TSCalculator. This component can be used as a stand-alone container that provides time series calculation features with <i>TSCalc</i> .
<code>com.jstatcom.parser</code>	
CalcEngine	A parser for the arithmetic language <i>TSCalc</i> .
CalcFunctions	Enumeration that holds all functions that can be used by CalcEngine. Can easily be extended with new function definitions.

Table 3.33: Elements of Calculator system

3.24.3 Architecture Background

The time series calculator is typically used as a ready made component that should be part of any application for time series analysis. It is integrated in the `TopFrame`, which will be described in detail in Section 3.25. Therefore, developers usually do not need to interact with the Calculator component directly, it is right in place by default. However, it is also possible to reuse either the `TSCalculatorPanel` or the `CalcEngine` alone. This could be useful if expressions in the TSCalc language are to be interpreted independently of the calculator component, or if a selection tool other than `TSSel` is to be used. Object composition gives a great deal of flexibility here to cope with various programming situations.

3.24.4 Usage Example

The use of the TSCalc language without the graphical calculator component is easily demonstrated with the following code snippet:

```
// init parser
CalcEngine parser = new CalcEngine();
// set a variable available for operations
parser.putVariable("s", new double[] { 123, 140, 110, 124});
// holds results
double[] result;
try {
    // parse a string in the TSCalc language
    parser.parseString("d = s|120;log_d=log(d )");
    // get the result
    result = parser.getVariable("log_d");
} catch (ParseException ex) {
    // deal with wrong input
    ex.printStackTrace();
}
```

The parser can be used to invoke various operations on number vectors that are relevant for time series analysis, but also for data based analysis in general.

3.24.5 Related View Packets

- Parent: View Packet 15: Time Series
- Children: none

3.25 View Packet 20: Components

3.25.1 Primary Presentation

System	Segment
Components	Application
	Data Table
	Equation

Table 3.34: Primary Presentation of Components

The Components system holds classes that are either themselves GUI components or that support user interface programming with JStatCom. The more self contained classes are directly part of the system, whereas more complex class structures are organized in subsystems. Table 3.34 shows the three child systems of which each contains several classes that together provide a certain type of functionality for GUI programming within the framework.

Figure 3.35 shows a class diagram of selected components that are all used for gathering and validating user input with a text field. The abstract class `IValTextField` provides a default implementation for the validating behaviour. Subclasses need to implement the actual validating algorithm, and must provide specific methods for the data type to be displayed. The `NumSelector` class can also be used to interact directly with the Symbol Management system to display and edit the value of a `JSCInt` or `JSCNumber` represented by an instance of `Symbol`. A screenshot of this often used component is given in Figure 3.36 after the user tried to specify a value outside the allowed range.

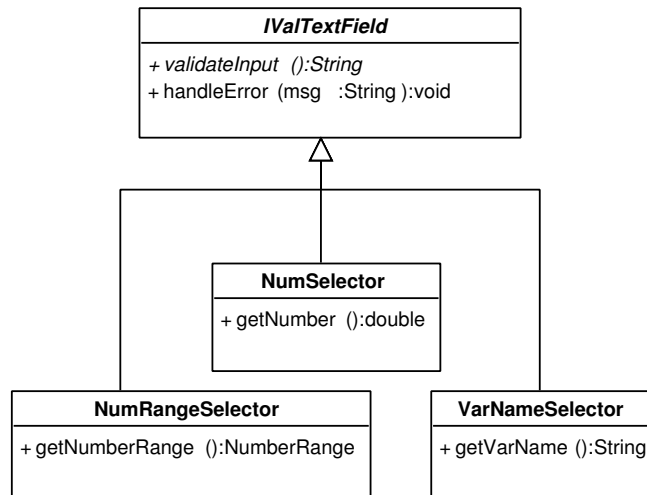


Figure 3.35: Classes for input validating text fields

It should be noted that also the classes `TSDateSelector` and `TSDateRangeSelector` from the Time Series system inherit the validating behaviour from `IValTextField`. This guarantees a consistent behaviour of all those components throughout the framework. In general, all classes of the Components system extend the behaviour of default Java Swing classes for the specific needs of data based analysis. This makes GUI development a lot faster and the resulting programs tend to contain fewer errors, because they can be build from well tested units. The use of these components is therefore strongly encouraged whenever appropriate.

3.25.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.component</code>	

<code>SymbolDisplay</code>	This text area displays the string value of a given data object defined by a symbol. It updates its display whenever the symbol changes.
<code>StdMessages</code>	Contains static methods to invoke simple confirm dialogs that are used to standardize messages of different types throughout JStatCom.
<code>IValTextField</code>	Abstract class that performs input validation before input focus is permanently moved away to ensure that no action is carried out, before input has been checked.
<code>NumSelector</code>	Subclass of <code>IValTextField</code> for number input that is automatically checked against a <code>NumberRange</code> object.
<code>NumRangeSelector</code>	Subclass of <code>IValTextField</code> for text input of a range specified by two komma separated numbers that is automatically checked against another <code>NumberRange</code> object.
<code>VarNameSelector</code>	Subclass of <code>IValTextField</code> to verify an input string against the rules for valid variable names throughout JStatCom.
<code>NumberFormatTypes</code>	Holds formatters that are used by JStatCom components to guarantee a consistent formatting behaviour.
<code>MultiLineLabel</code>	Label component that can display text with more than one line.
<code>MatrixBorder</code>	This border puts angular brackets [] around a component.

<code>AutoEnableMenu</code>	A <code>JMenu</code> that gets enabled if at least one of its menu items becomes enabled.
<code>OutHolder</code>	Interface to be implemented by components that hold text output assembled by <code>PCall</code> objects..
<code>SystemOutHolder</code>	Simple implementation of the <code>OutHolder</code> interface that redirects output to <code>System.out</code> .
<code>ResultField</code>	Implementation of <code>OutHolder</code> that provides a scrollable text area for presenting output in text form.
<code>ResultFieldPopup</code>	A popup menu especially designed to work with the <code>ResultField</code> component. It can be used to clear, print, and save the contained text.
<code>Card</code>	Interface to be implemented by components that are used together with <code>CardDisplayPanel</code> and that should be informed when they are shown or hidden.
<code>CardDisplayPanel</code>	An alternative to the standard Swing <code>CardPanel</code> that can be used in conjunction with <code>CardPanelAction</code> to display components on demand as well as to notify them when they are either shown or hidden.
<code>CardPanelAction</code>	An action for the task of showing a component in a <code>CardDisplayPanel</code> . Can be used to create <code>Card</code> instances on demand from their class name.
<code>CheckBoxList</code>	A special <code>JList</code> that renders its items as selectable <code>JCheckBox</code> components. Useful for selection of boolean properties for a dynamically changing set of objects.

CompSettings	Specifies settings for components used throughout JStatCom for consistency.
subsystems	
Application	Holds the main application frame and related classes with useful default functionality, like data import, time series calculator, modules management, help system, status display, logging facility, and the symbol control frame. It can be configured with external property files or via subclassing.
Data Table	Subsystem to display the array data objects JSCNArray and JSCSArray in tables with various editing and display options.
Equation	Subsystem to represent dynamic equations graphically with their parameters and variable names.

Table 3.35: Elements of Components system

3.25.3 Context

The Components system contains many classes that users directly interact with. They are the general building blocks of the graphical user interfaces for scientific models. More specific components are found in respective subsystems, like for example in the Time Series system. Figure 3.37 shows the uses of the Components system for two different actors, users and developers. The use cases for the user define the type of components that are needed. GUI developers use the system typically with visual programming tools to effectively layout user interfaces.

3.25.4 Architecture Background

Component-based development is one of the current paradigms in software development (Pree (1997)). The distinction between a component and a class is not very

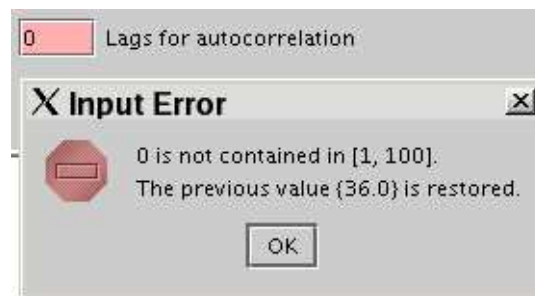


Figure 3.36: Screenshot of NumSelector with an error message

sharp, all elements of the Components system are Java classes. However, they conform to the JavaBeans specification, which allows to visually compose them into composite components and applications using graphical application builder tools. Following Hallway (2002), the definition of a *component* is given as an independent unit of production and deployment that is combined with other components to assemble an application. JStatCom uses this concept in various ways to support a modern approach to programming.

It is not only the preferred way to build the GUI elements of an application, but with the Engine system it also treats algorithms written in other languages as well as external engines as components. The advantage of using a component architecture is that one can reuse and combine independent binary units to develop applications much more rapidly than if the respective features would need to be reimplemented. Reusability is therefore achieved not only for classes of the Java language, but also for other software units that provide key functionality for the given problem domain. Typical problems of this approach are related to deployment of those binaries, possible version conflicts and interface incompatibilities. Especially the Application subsystem uses a component approach to load independent modules via definitions set in a configuration file, which will be described in greater detail in Section 3.26.

3.25.5 Usage Example

As a representative example, Figure 3.38 shows the specification of a component for number selection in a visual GUI builder. First, the component was selected

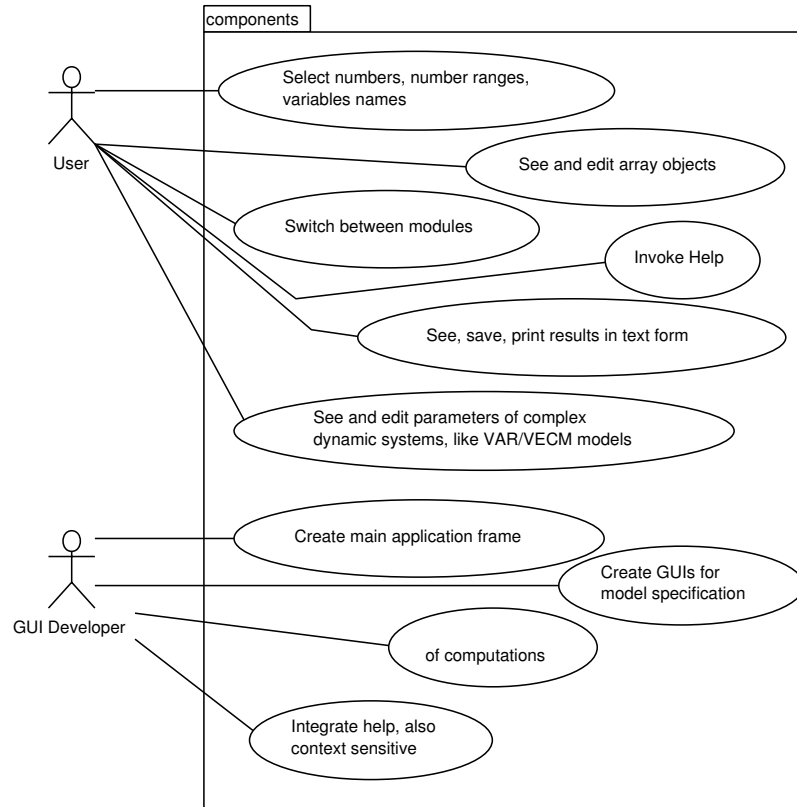


Figure 3.37: Use cases for the Components system

from a list of available beans and placed on a `JPanel`, which is rendered as the gray area. The property editor to the right can then be used to easily configure the behaviour of that component. Many options are displayed, but mostly only very few need to be changed from their default. Some of the specific options for the `NumSelector` are highlighted with a red cross in the graphic. The initial number is set via the `number` property, the allowed range can directly be given via the `rangeExpr` or `numberRange` property, where `rangeExpr` allows to input the range intuitively as a string, instead of using a reference to a `NumberRange` object. The setting `intType` specifies whether the resulting number would be stored as a `JSCInt` or a `JSCNumber` instance in the symbol table. However, in the current setting this option has no effect, because the component does not store its state in

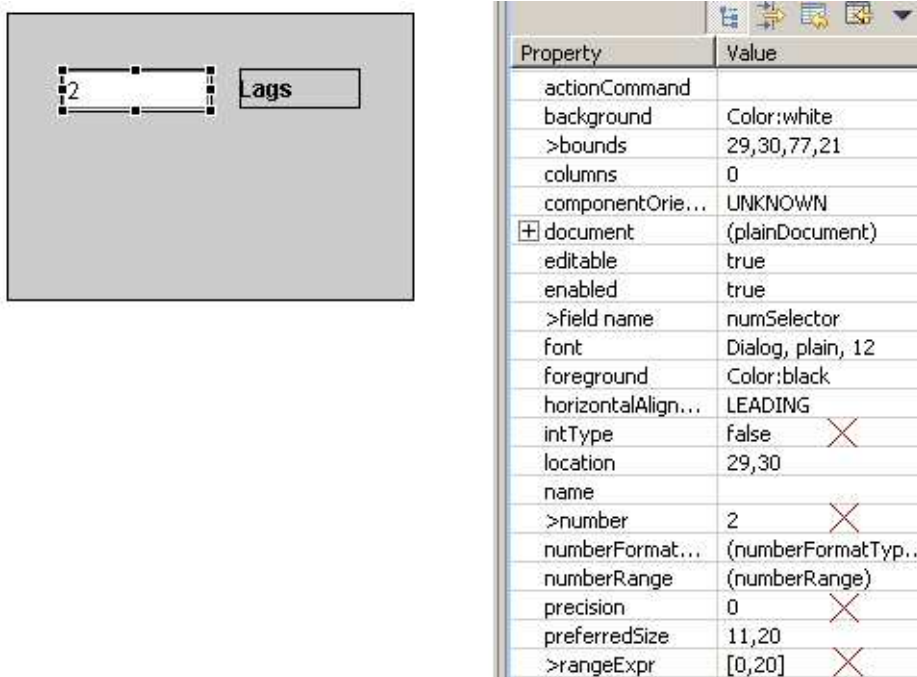


Figure 3.38: Specification of a NumSelector in a visual application builder

a symbol. For this to happen, the `symbolName` option would have to be set, which is not shown in the picture, but appears further down in the property sheet. All other options are left unchanged for the example. It is also possible to program this manually. The following piece of code shows what is actually generated by the builder tool:

```
// initialization method for a number selector
private NumSelector getNumSelector() {
    if (numSelector == null) {
        numSelector = new NumSelector();
        numSelector.setBounds(29, 30, 77, 21);
        numSelector.setRangeExpr("[0,20]");
        numSelector.setNumber(2.0);
    }
    return numSelector;
}
```

This method is part of a class that subclasses `JPanel` here and is invoked

on initialization. The rest of the code is omitted for clarity. In a similar way, all graphical components in JStatCom can be used and configured. Typically the GUI builder tool is employed to do the initial layout. Later changes are then often done by hand, especially if they do not affect how the component is placed and sized.

3.25.6 Related View Packets

- Parent: View Packet 1: JStatCom

- Children:
 - View Packet 21: Application
 - View Packet 22: Data Table
 - View Packet 23: Equation

3.26 View Packet 21: Application

3.26.1 Primary Presentation

The Application subsystem is one of the entry points for developers using JStatCom. Figure 3.39 shows the participating classes. It should be mentioned that the classes in the lower part of the diagram are elements of other subsystems. This reflects the main task of the Application system, namely providing a ready made frame with access to various features of the framework. GUI developers can reuse this to easily embed newly created modules without having to worry about data import, modules management, help system access, symbol control, status display of engine requests, access to the time series calculator, and logging. The system also provides a default splash screen and about information which can be configured easily for the respective application. In later stages also project management will be integrated on the framework level. The main application frame can be seen in Figure 3.40. It has been customized for the software JMulTi and several modules have been loaded. They are available via the selected menu and are displayed with their name and an icon.

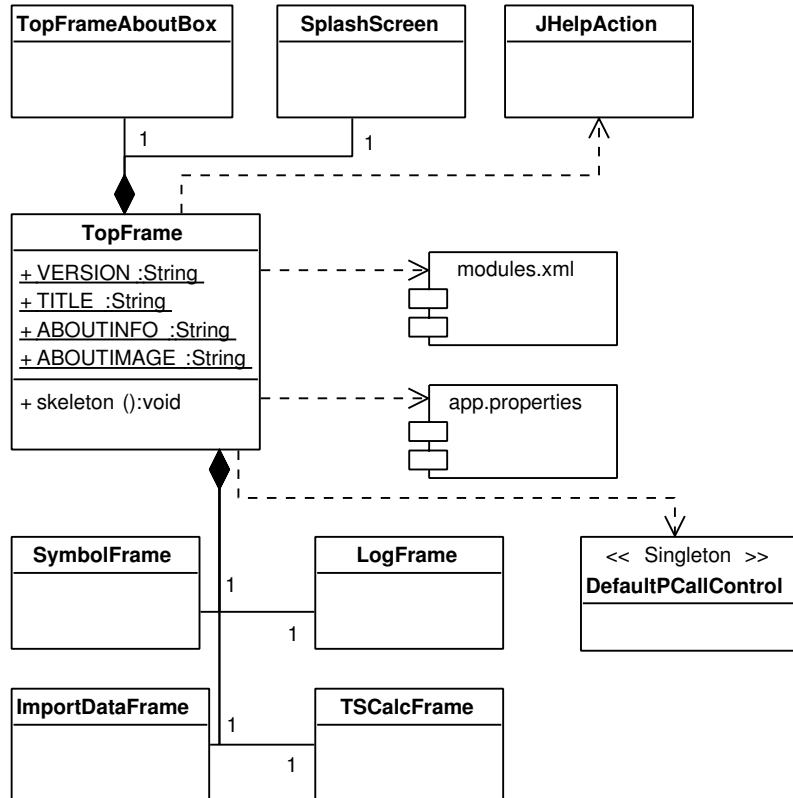


Figure 3.39: Classes of Application system

With the Application system, GUI developers can concentrate on creating modules for specific science models without having to worry about the general infrastructure of their application. Moreover, the system standardizes the appearance of applications based on JStatCom and reduces the need for ad hoc solutions for recurring tasks. However, it is also obvious that this system is likely to be changed or extended by developers. Therefore very likely changes, like the image for the splash screen, the title of the application, or the about information, can be done by just editing the property file `app.properties`. A very important feature is to easily integrate new modules. This can be done by editing the file `modules.xml`. This way, typical adjustments can be done without changing the Java code at all.

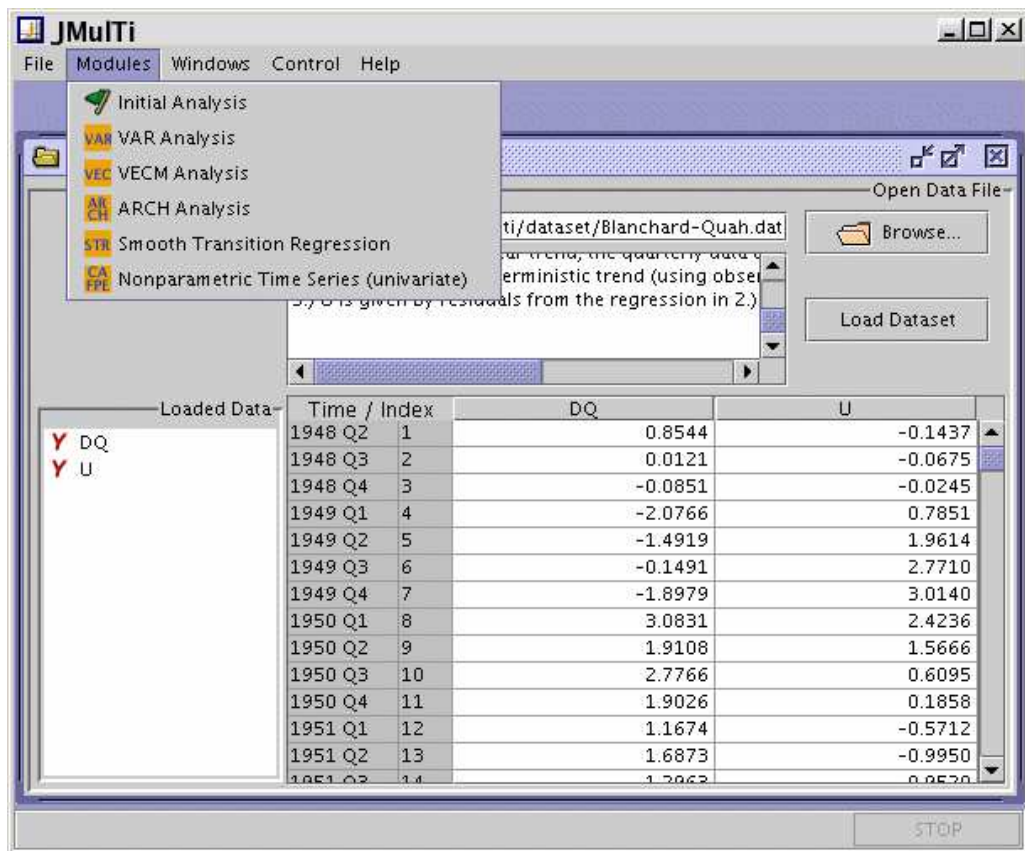


Figure 3.40: Screenshot of TopFrame customized for the JMulti application

To start the application, calling the main method of TopFrame is required. For convenience, this can easily be done with the `app.bat/app.sh` script, which also needs to set the classpath for all modules that are to be loaded.

Resources needed by the Application system

The mentioned configuration files are the resources that are required by the Application system. One should notice that the `app` scripts to start the program are useful for development because they can easily be adjusted. But for the final deployment version this way of starting a Java application is not state-of-the-art anymore. One should rather create a jar archive with a manifest file that points to the class containing the main method. For the default case this would be

`com.jstatcom.component.TopFrame`. There is a file `MANIFEST.mf` with these settings in the working directory of `JStatCom`. The jar archive should be created with the ANT tool using the settings in the file `build.xml`. Typically this can easily be invoked from within an IDE. The user could then start the application by simply double clicking on that jar archive without having a console window appearing. Another way of starting the program would be to create a system specific executable with the help of external tools, but this is not discussed here.

Resource	Usage
files in working directory of <code>JStatCom</code>	
<code>modules.xml</code>	XML file that holds definitions for all modules that should be loaded by the Application system.
<code>help_context.xml</code>	XML file to set the help IDs for context-sensitive help that can automatically be displayed for certain components.
<code>app.bat</code> (Windows) <code>app.sh</code> (Linux/Unix)	Helper script to start the application by invoking the <code>main</code> method of <code>TopFrame</code> . The classpath might need to be adjusted to include all required classes.
<code>app.properties</code>	Property file to set the version, the title, the logo image, and the about information for an application based on <code>JStatCom</code> .

Table 3.36: Resources for the Application system

3.26.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.component</code>	
<code>TopFrame</code>	A configurable component that should be used as the main frame for applications based on JStatCom. Provides default functionality for loading data, the module management, the time series calculator, a logging window, the symbol control frame and a status display.
<code>TopFrameReference</code>	Holds a globally accessible reference to the top frame of the application. Needed by modal dialogs to appear on top of the application frame.
<code>TopFrameAboutBox</code>	Default dialog with information about the software.
<code>Module</code>	Represents a module to be used with <code>TopFrame</code> . Helper class to enable module configuration via an XML file.
<code>SplashScreen</code>	A windows that is shown when starting applications based on JStatCom.
<code>JHelpAction</code>	Helper class that facilitates use of the JavaHelp system.
<code>JHelpContextMgr</code>	Manages the mapping between IDs for context-sensitive help topics and the respective components.

Table 3.37: Elements of Application system

3.26.3 Architecture Background

The Application system was developed to provide easy access to many features of JStatCom without the need to write a lot of code. Because the basic idea of using a framework is to reuse a general design, the TopFrame component was created to relieve the developer from the burden of always starting from scratch when designing new modules. This concept is not new and can for example be seen in the GiveWin/OxPack econometric software package as well. Developers only need to implement a subclass of ModelBase, which is then automatically integrated in an application framework with rich default functionality, like reading in data and project management. The same pattern can often be found in framework approaches, like for example in the Eclipse project, where a standard application frame can be extended with plug-in modules. Thus, module developers can focus on specific problems in a given context.

To support this, a true component based programming approach is required, because the Application system has to gather information about which modules to load and about its own appearance from external resources. This information is not known at compile time, but is only retrieved when the program is started. The emphasis of the runtime context is typical of component development. Therefore the module extension mechanism must make certain assumptions about the type of the modules to load and must also provide ways to deal with missing or wrong modules. Because it does not know about the modules at compile time, many checks done by the compiler are bypassed and have to be done at runtime instead.

Defining Modules

A module for JStatCom is just defined as a subclass of the standard Swing class `javax.swing.JInternalFrame`, nothing more. This is a very general definition that does not make any further assumptions, than that the module is a component that can be displayed in a desktop panel. But typically, modules are subclasses of `ModelFrame`, which is itself a subclass of `JInternalFrame`, because it sets up the global symbol table for child panels as described in Section 3.10. A special case of a module is just a helpset. In this case, no component is displayed but the specified `JavaHelp` set is dynamically loaded at program start and can be accessed

via the help system together with all other helpsets. To load modules via the module extension mechanism, a new entry in the file `modules.xml` should be made. This way, no Java code is required to set up new menu items and actions. It is also not required to load the corresponding helpset manually. The module definition file is written in the XML language, the de facto standard for structured documents. The advantage of using XML is that standard parsers can be reused and that it is easy to retrieve the desired information from such a document. The only drawback might be that it looks slightly less familiar than simpler text files for settings. However, it bears similarities to HTML and is getting more and more common. An example module definition file could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<TopFrame$XMLRep xmlns="java:com.jstatcom.component" name="JMulti">
  <Module helpset="helpset/jstatcom/jstatcom.hs"/>
  <Module class="de.jmulti.initanal.InitAnalFrame"
    helpset="helpset/initanal/InitAnal.hs"/>

  <Module class="de.jmulti.vecm.VECM"
    helpset="helpset/vecmodels/VECM.hs"
    method="getInternalFrame"/>
</TopFrame$XMLRep>
```

Each module must be put in a new tag `Module` with several possible attributes that can take on certain values. Each value is defined as a string that must be put in double quotes. Attributes can be used as follows:

- **class** - If the module is not just a helpset, this attribute takes the fully qualified classname of the module to load. The class must be a subclass of `javax.swing.JInternalFrame`. If the `method` property is not set, then the default constructor is used to create an instance of the given class. The class must be in the classpath of the application, otherwise it would not be found by the class loader.
- **method** - Sometimes modules provide a certain static initializer method that returns an instance of the underlying class. If such a method should

be called instead of the default constructor, this attribute has to be set with the method name. The method to be called must be static, it must take no arguments, and it must return an instance of the module class. This attribute is ignored if no `class` attribute is set.

- **helpset** - Can be used to set the helpset that is loaded dynamically via the JavaHelp system. It will be merged with the other helpsets at program start. It can also be used without a `class` attribute. The helpset must be in the JavaHelp format and it has to be located in the resource directory of the application. Typically it is part of the jar archive of the application, or it is in a subdirectory of the working directory.

Setting Properties

Some small adjustments can be made in the file `app.properties`, which is in the default Java property format. It is used to define some information about the respective program that can easily be changed without modifying the Java code. The values for each setting must be put on one line, even if it might become very long. Linebreaks must be indicated with a `\n`. The default settings are:

```
VERSION = 1.0
TITLE = JStatCom
ABOUTIMAGE = /images/splashdefault.gif
ABOUTINFO = www.jstatcom.com\n\nA. Benkwitz & M. Krätzig 2000-2004
```

- **VERSION** - Holds a string describing the release. It appears in the splash screen at program start and in the about box.
- **TITLE** - Holds the title of the application which appears in the top level frame.
- **ABOUTIMAGE** - Points to an image that is displayed in the splash screen and in the about box. The suggested size is 350 x 250 px.
- **ABOUTINFO** - String containing author and copyright information, as well as credits, project homepage, etc.

Managing Context-Sensitive Help

Modern help systems offer the possibility to easily navigate to a help topic that describes a certain component by just clicking with the mouse on the GUI when it is in context-sensitive help mode. If a so called help ID has been set before for that component, the corresponding topic is displayed. Otherwise, the default help page is shown, typically the one that was last visited. Of course it helps users if IDs for many components are set. JStatCom offers a way to maintain the mapping between components and IDs in an external XML file `help_context.xml`. This has the advantage that help IDs can be set in a single place, and that no Java code has to be changed when IDs change or components are added to the application. The class that uses this file and manages the components and IDs is the singleton `JHelpContextMgr`. Every component that wants to use this service needs to register with that class. However, to further simplify the required coding, all instances of `ModelPanel` are automatically registered with the help context manager on object construction. Typically, context-sensitive help is required only for subclasses of `ModelPanel`, therefore this basically eliminates the need to register components manually, except for special cases in which other classes are used.

The only step that has to be done manually is to edit the file `help_context.xml`. Each component is identified with its class name and takes the help ID as the target for the help topic to display. Help IDs are defined in JavaHelp sets and can be retrieved easily with the tool `JHelpDev`, see Appendix B. An example context definition file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<JHelpContextMgr xmlns="java:com.jstatcom.component">
  <JHelpContextMgr$ContextID class="com.jstatcom.io.TSImportPanel"
    helpid="jmulti.impdata"/>
  <JHelpContextMgr$ContextID class="de.jmulti.var.VAREstCoeffPanel"
    helpid="varmodels.node2"/>
  <JHelpContextMgr$ContextID class="de.jmulti.var.VARSubSetPanel"
    helpid="varmodels.node4"/>
</JHelpContextMgr>
```

The format is again XML. Each entry must start with the string `JHelpContextMgr$ContextID` and can take the following attributes:

- **class** - This attribute takes the fully qualified classname of the component that corresponds to a help topic. Typically, classes are of type `ModelPanel`, which are the containers that hold input specification GUIs for calling scientific procedures.
- **helpid** - The help ID that defines the topic to be displayed when the mouse is clicked on an instance of the class that has been set. The mouse has to be in context-sensitive help mode, which is a special option in the default help menu of `TopFrame`.

Subclassing `TopFrame`

Much emphasis has been put on being able to do likely changes of the Application system without the need to write any Java code. However, if more customization is required, like adding new menus or menu items to the default frame, subclassing `TopFrame` is necessary. Most methods can be overwritten. Also, there is a special method `extra` which is empty by default and could be implemented by subclasses to add additional features. To initialize the subclass of `TopFrame`, the method `skeleton` needs to be called. A code example is given in the usage section.

3.26.4 Usage Example

This example shows, how one could subclass `TopFrame` to put a new item in the *Control* menu. This should invoke a panel to configure plot settings for the Gauss engine system. It is not part of the default implementation, because it assumes the availability of that engine. When `TopFrame` is overwritten, the new class should be invoked instead. Therefore, in the `app` script the class name should be replaced with the name of the subclass, which is `MyApp`. Otherwise, the changes would have no effect, because the new class would never be used. All other configurations options that have been described in the previous section are still used, unless the respective methods are directly overwritten.

The code shows the whole class `MyApp` that inherits from `TopFrame`. The method `extra` is empty in the superclass and can safely be overwritten to add special configurations. Here it is used to create a new item with an attached action

that is then added to the *Control* menu. It is also important to provide a static `main` method, which creates an instance of the newly defined class and calls the `skeleton` method of the superclass. Without this, the `main` method of `TopFrame` would be called, which only creates a `TopFrame` instance and no changes would be visible.

```
// subclass TopFrame
public final class MyApp extends TopFrame {
    // empty superclass method is overwritten
    protected void extra() {
        // create new menu item
        JMenuItem item = new JMenuItem("Global GAUSS Graphics Settings");
        item.addActionListener(
            new AbstractAction("Global GAUSS Graphics Settings") {
                public void actionPerformed(ActionEvent evt) {
                    ... invoke plot configuration dialog
                }
            });
        // add item to control menu from superclass at position 0
        getControlMenu().add(item, 0);
    }
    // method that starts the application
    public static void main(String[] args) {
        try {
            MyApp myFrame = new MyApp();
            // call skeleton
            myFrame.skeleton();
        } catch (Throwable exception) {
            System.err.println("Exception occurred in main()");
            exception.printStackTrace();
        }
    }
}
```

For a detailed documentation of all methods that can be overwritten, the API documentation of `TopFrame` should be used.

3.26.5 Related View Packets

- Parent: View Packet 20: Components
- Children: none

3.27 View Packet 22: Data Table

3.27.1 Primary Presentation

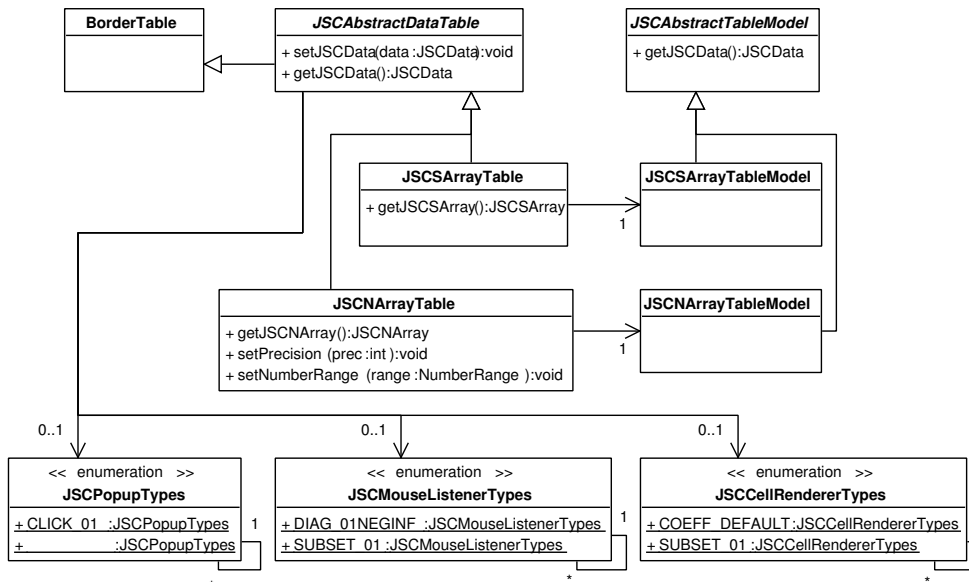


Figure 3.41: Classes of Data Table system

The Data Table system can be applied to display and edit array data objects that are part of the Type System of JStatCom. Currently, the data types representing arrays are JSCNArray and JSCSArray. The system extends the functionality of the default Java Swing table components to meet the special needs for rendering and modifying number and string arrays.

Figure 3.41 shows the classes which are part of the Data Table system. A general class `BorderTable` is used for the purely technical reason to let tables

	1	2	3	4	5	6
1	1.8252	-0.3994	3.5981	-0.7873	0.2666	-0.0583
2	-0.3994	0.2032	-0.7873	0.4006	-0.0583	0.0297
3	3.5981	-0.7873	14.8197	-3.2427	2.4985	-0.5467
4	-0.7873	0.4006	-3.2427	1.6498	-0.5467	0.2781
5	0.2666	-0.0583	2.4985	-0.5467	1.2997	-0.2844
6	-0.0583	0.0297	-0.5467	0.2781	-0.2844	0.1447
7	-3.8232	0.8365	-14.8683	3.2533	-2.4427	0.5345
8	0.8365	-0.4256	3.2533	-1.6552	0.5345	-0.2719
9	0.0599	-0.0131	0.1952	-0.0427	0.0352	-0.0077

Figure 3.42: Screenshot of a NArrayTable

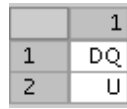
e	prod	rw	U
1	0	0	0
0	1	0	1
*	0	1	*
0	1	0	1

Figure 3.43: Screenshot of NArrayTable with a special table renderer and mouse click listener

respect borders, which is not the case for the default `JTable`. The abstract class `JSCAbstractDataTable` manages the general functionality of all potential data tables, but it must be subclassed by concrete implementations for the respective array types. The two implementations that are available are `JSCNArrayTable` and `JSCSArrayTable`. Each of them has special features for the data type to display.

In a similar way, the underlying table model is first defined in an abstract class `JSCAbstractTableModel` and then further specified in the concrete models `JSCSArrayTableModel` and `JSCNArrayTableModel`, which are used with their respective tables. The distinction between a component for view and control and a class representing the underlying model follows standard Swing patterns and is an example of Model-View-Controller design.

A special feature of the Data Table system is that it provides great flexibility to adjust the view and controller behaviour of tables. The view can be changed by setting specific cell renderers that define how values are actually displayed on screen. It can be set via the method `setCellRenderer` of



	1
1	DQ
Z	U

Figure 3.44: Screenshot of a SArrayTable

JSCAbstractDataTable. The class that helps to define those renderers and that also provides an enumeration of specified cell renderers is

JSCCellRendererTypes. For example, it might be useful to display number values not as digits, but as symbols that mark special values. This might be required for tables that represent restriction matrices that only take zeros and ones, for example. The renderer can also be used to change the background color of some cells to indicate that they cannot be edited. Figures 3.42 and 3.43 show screenshots of tables for number arrays with different renderers. In Figure 3.42, the default renderer for numbers is used with a number precision of 4 digits. Figure 3.43 displays a table that uses a special renderer that displays '∞', '1' and '0' for the values Infinity, 1, and 0 respectively. The values zero and one indicate restrictions on the underlying coefficients, whereas Infinity is a code for an unrestricted coefficient that has to be estimated. Figure 3.44 presents a screenshot of a JSCSArrayTable. Typically most configuration options for tables are used only for number tables, because the default behaviour of string array tables does not need to be changed often.

The controller defines how values represented by the underlying model can be changed. The standard way of modifying those values would be to edit the values directly in their cells with a text editor component. This is the default behaviour if tables are set to be editable. However, sometimes it is much more convenient to change many or all values of a table simultaneously, or to just switch between a given set of values that cells can take. The Data Table system provides two classes to support these adjustments.

JSCPopupTypes is a class that can be used to define popup menus that invoke actions on the underlying table. It also contains an enumeration of already implemented popup menus for the default case and for special purposes. One can set the menu to be used via the method `setTablePopup` of `JSCAbstractDataTable`.

The default menu for number tables can be used to change the precision of the displayed numbers. One can also switch between exponential notation and decimal notation. If this menu is used, the underlying values are not changed at all, but just the way how they are rendered. It is also possible to use a popup menu that sets all data cells of the underlying table to certain values instead. This could be useful for tables representing restriction matrices, if one wants to change all values efficiently with a single mouse click. The popup can also be removed completely by setting it to `null`.

Another class that might be employed to change the controller behaviour of data tables is `JSCMouseListenerTypes`, which can be set via `setMouseListener` of `JSCAbstractDataTable`. As is indicated by the name, this class helps to specify mouse listeners. Like the other enumerations, it defines specified listeners that can be reused. Mouse listeners allow to attach an action to a certain mouse behaviour. For number tables, the most common action would be to change a value to the next valid one from a given set, when the mouse is clicked on a cell. A typical set of values is `{0, 1}`. By clicking on the cells, the displayed values would just switch between those two values. It is important to note that the table should not be editable in this case, because then the mouse click would also bring up a text editor and the editing behaviour would be somewhat strange. By setting the mouse listener to `null`, no specific action would be attached to the underlying table.

Whenever data tables are used, they should be put in a scrollpane container. This makes sure that all values can be accessed, even if the data table gets too large to be displayed completely. In this case, scrollbars would appear. A special scrollpane for the Data Table system is provided via `JSCDataTableScrollPane`. It is recommended to use this class instead of the default `JScrollPane`, because it provides some helpful methods for displaying row and column headers, as can be seen in Figures 3.42, 3.43, and 3.44. It also adjusts its size and the position of the scrollbars differently than the default scrollpane, and it allows to specify a maximum number of columns and rows to be displayed before starting to scroll. However, the use of that component is by no means mandatory.

The Data Table system works closely together with the Type System and the Symbol Management. Data tables can display the values represented by symbols

from a symbol table. In the same way, row headers and column headers of data table scrollpanes can show values from symbol tables. This way, data tables can be used to display values stored in symbols, and they update automatically whenever the values of those symbols change. In the same way, they can be used to change the values of the underlying symbols via their controller behaviour. Therefore this system provides very efficient ways of interacting with array data. Programming with it simplifies otherwise complex user interaction.

3.27.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.table</code>	
<code>NumberDisplayTable</code>	Interface for components that display numbers and that can be configured how the value should be printed.
<code>BorderTable</code>	Subclass of <code>JTable</code> that respects the border set.
<code>BorderTableBorder</code>	Most inner border for <code>BorderTable</code> .
<code>JSCAbstractDataTable</code>	Abstract table to be subclassed by tables for data arrays, like <code>JSCSArray</code> and <code>JSCNArray</code> .
<code>JSCAbstractTableModel</code>	A table model to be used with <code>JSCAbstractDataTable</code> instances.
<code>JSCCellRendererTypes</code>	Enumeration with different cell renderers for the <code>JSCAbstractDataTable</code> which can be used in different modeling contexts, for example to display coefficients matrices or subset matrices.
<code>JSCDataTable-ScrollPane</code>	Special scrollpane to be used with data tables.

JSCDataTable-ScrollPaneLayout	A ScrollPaneLayout to be used with the JSCDataTableScrollPane.
JSCMouseListenerTypes	A MouseListener to be used with a JSCAbstractDataTable that reacts to mousePressed events and double clicks.
JSCNArrayTable	Implementation of JSCAbstractDataTable for JSCNArray data objects.
JSCNArrayTableModel	Implementation of JSCAbstractTableModel for JSCNArray data objects.
JSCPopupTypes	Enumeration of different popups that can be used by data tables.
JSCSArrayTable	Implementation of JSCAbstractDataTable for JSCSArray data objects.
JSCSArrayTableModel	Implementation of JSCAbstractTableModel for JSCSArray data objects.

Table 3.38: Elements of Data Table system

3.27.3 Architecture Background

Like most parts of JStatCom, the Data Table system was designed with flexibility and extendibility in mind. Flexibility was achieved with providing various ways of customizing display and editing behaviour of data tables. This was a requirement especially for number arrays, because they are used in various different situations to specify input, as well as to show results.

Extendibility means that all options can be expanded with further implementations to meet special needs that cannot be anticipated. For example, the enumerations `JSCMouseListenerTypes`, `JSCPopupTypes`, and `JSCCellRendererTypes` can be subclassed to define new types with the behaviour required for a spe-

cial situation. Furthermore, it means that the Data Table system could work together also with new data types that might be added in the future without changing its general design. A new table implementation would require to subclass `JSCAbstractDataTable`, as well as `JSCAbstractTableModel`.

3.27.4 Usage Example

The Data Table components `JSCSArrayTable`, `JSCNArrayTable`, and `JSCDataTableScrollPane` can efficiently be used with graphical builder tools. It is also possible to code “by hand”, but especially layout issues are tedious to handle that way. The following steps are typical:

1. Create some container, most often a `ModelPanel`, or simply a `JPanel`.
2. Place a `JSCDataTableScrollPane` on top of it.
3. Choose a `JSCSArrayTable` or `JSCNArrayTable` and place it inside that scrollpane. The choice of the table depends on whether a number or a string array should be displayed.
4. Adjust special settings for the scrollpane and for the data table via the property editor provided by the builder tool.

The code that is generated could look like the following. The resulting component actually corresponds to Figure 3.43.

```
private JSCNArrayTable dataTableB = null;
private JSCDataTableScrollPane dataTableScrollPaneB = null;
private ModelPanel panel = null;
...
// initializes data table
private JSCNArrayTable getDataTableB() {
    if (dataTableB == null) {
        dataTableB = new JSCNArrayTable();
        dataTableB.setCellRenderer(JSCCellRendererTypes.DIAG_01M1NEGINF);
        dataTableB.setTablePopup(null);
        dataTableB.addMouseListener(JSCMouseListenerTypes.DIAG_0NEGINF);
    }
}
```

```

        dataTableB.setEditable(false);
        dataTableB.setSymbolName("B_MATRIX");
        return dataTableB;
    }

    // initializes scrollpane
    private JSCDataTableScrollPane getDataTableScrollPaneB() {
        if (dataTableScrollPaneB == null) {
            dataTableScrollPaneB = new JSCDataTableScrollPane();
            dataTableScrollPaneB.setColumnHeaderShowing(true);
            dataTableScrollPaneB.setMinimumVisibleColumns(5);
            dataTableScrollPaneB.setMinimumVisibleRows(5);
            dataTableScrollPaneB.setViewportView(getDataTableB());
            dataTableScrollPaneB.setColumnHeaderSymbolName("Ny");
            return dataTableScrollPaneB;
        }
        // initializes component, adds scrollpane with table to panel
        private initialize(){
            ...
            panel.add(getDataTableScrollPaneB());
            ...
        }
    }

```

In the method `getDataTableB` a number array table is created and configured. A special cell renderer is set, the table popup is removed, and a mouse listener is installed. For more details on the used types, the API documentation should be consulted. Furthermore, the table is set to non editable, meaning that it can only be changed via the installed mouse listener. Finally, a symbol name is set, which is retrieved from the global symbol table. The table displays the values stored there, and changes in the table are written to that symbol.

The method `getDataTableScrollPaneB` sets up the enclosing scrollpane with a column header. A minimum of 5 rows and columns will be displayed before starting to scroll. This affects the minimum size required by this component. Lastly, the symbol name to be displayed in the column header is set. It must point to a symbol of type `SARRAY`. In the example, it holds the names of the selected

endogenous variables, as can be seen in Figure 3.43. In the `initialize` method, the scrollpane is added to the underlying container, which is a `ModelPanel` here. It should be noted that typically there is more code for layout and for other components, like buttons, text fields, etc. This is omitted here for clarity.

3.27.5 Related View Packets

- Parent: View Packet 20: Components
- Children: none

3.28 View Packet 23: Equation

3.28.1 Primary Presentation

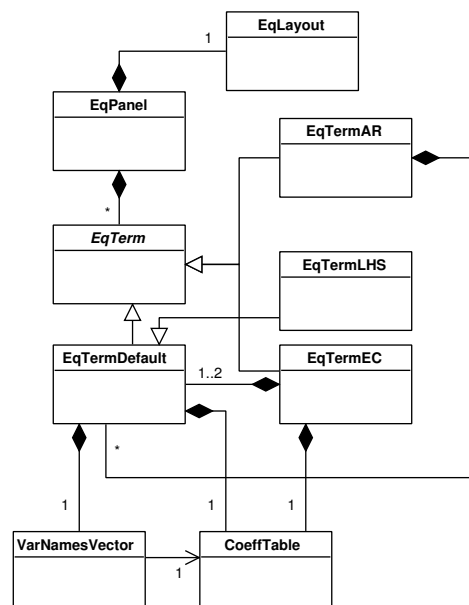


Figure 3.45: Classes of Equation system

$$\begin{bmatrix} d(\text{gnp})(t) \\ d(m)(t) \\ d(Dp)(t) \end{bmatrix} = \begin{bmatrix} 0.484 \\ 0.184 \\ 0.382 \end{bmatrix} \begin{bmatrix} 1.000 & -0.790 & -9.816 \end{bmatrix} \begin{bmatrix} \text{gnp}(t-1) \\ m(t-1) \\ Dp(t-1) \end{bmatrix} + \begin{bmatrix} -0.615 & -0.030 & 2.831 \\ -0.156 & -0.003 & 1.149 \\ -0.224 & -0.034 & 1.835 \end{bmatrix} \begin{bmatrix} d(\text{gnp})(t-1) \\ d(m)(t-1) \\ d(Dp)(t-1) \end{bmatrix} \\
 + \begin{bmatrix} -0.441 & 0.306 & 1.442 \\ 0.219 & -0.231 & 0.457 \\ -0.164 & 0.035 & 0.951 \end{bmatrix} \begin{bmatrix} d(\text{gnp})(t-2) \\ d(m)(t-2) \\ d(Dp)(t-2) \end{bmatrix} + \begin{bmatrix} -0.321 \\ -0.114 \\ -0.259 \end{bmatrix} \text{CONST} + \begin{bmatrix} u1(t) \\ u2(t) \\ u3(t) \end{bmatrix}$$

Figure 3.46: Screenshot of VEC model coefficients estimates

$$\begin{bmatrix} d(\text{gnp})(t) \\ d(m)(t) \\ d(Dp)(t) \end{bmatrix} = \begin{bmatrix} * \\ * \\ * \end{bmatrix} \text{ec1}(t-1) + \begin{bmatrix} * & 0 & * \\ * & 0 & * \\ * & 0 & * \end{bmatrix} \begin{bmatrix} d(\text{gnp})(t-1) \\ d(m)(t-1) \\ d(Dp)(t-1) \end{bmatrix} + \begin{bmatrix} * & * & * \\ * & * & * \\ * & 0 & * \end{bmatrix} \begin{bmatrix} d(\text{gnp})(t-2) \\ d(m)(t-2) \\ d(Dp)(t-2) \end{bmatrix}$$

Figure 3.47: Screenshot of VEC model exclusion restrictions on shortrun dynamics

For representing dynamic models intuitively, JStatCom provides the Equation subsystem. It can be used to display (V)AR and VEC models, but also simple equations or systems of equations. Figures 3.46 and 3.47 show screenshots of equations at runtime. The classes of that system are presented in Figure 3.45. EqPanel is a subclass of the standard JPanel with a special layout manager EqLayout that manages the display of potentially large equations with many terms. Therefore, EqPanel should always be used as a container for equation terms instead of JPanel, especially when the number of displayed parts varies dynamically.

The equations can be build from several available equation terms that all inherit from the abstract class EqTerm. The class EqTermDefault is a simple equation term that has one instance of VarNamesVector and one CoeffTable. The variable names vector is a subclass of JSCSArrayTable, whereas the coefficients table subclasses JSCNArrayTable. Each default equation term displays variable names together with the corresponding coefficients. It can also be adjusted to only show the variable names.

The class `EqTermDefault` can also be used to compose more complex equation terms for special situations. So far, the following classes are implemented:

1. `EqTermAR` - An equation term that can have many default terms, each of them representing lagged variables and the corresponding coefficients. The starting lag can be adjusted.
2. `EqTermEC` - A special equation term to represent the long term equilibrium in a VEC model. It has up to two default terms, one for endogenous variables, and one for the restricted deterministics. It also has another `CoeffTable` for the loading parameters. It can be seen in Figure 3.46.
3. `EqTermLHS` - The equation term that should be used on the left hand side of an equation. Typically, it only displays the endogenous variables, but it might also display structural coefficients as well. It inherits from `EqTermDefault` to overwrite some standard settings.

The Equation system is build on top of the Data Table system, because it needs to display arrays of numbers and strings. Therefore, it also provides great flexibility to set special renderers, mouse listeners, and popup menus. Figure 3.47 shows an equation with a renderer that displays symbols for special values. It also has a mouse listener attached that switches between *restricted* and *unrestricted* if the mouse clicks on a cell. A popup menu can be used to set whole matrices to a certain value for more efficient editing.

3.28.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.equation</code>	
<code>CoeffTable</code>	Subclass of <code>JSCNArrayTable</code> to display coefficients of a model.
<code>EqLayout</code>	Layout to be used with <code>EqPanel</code> .

<code>EqPanel</code>	Panel to be used as the container for all <code>EqTerm</code> components.
<code>EqTerm</code>	Abstract superclass for all equation term components.
<code>EqTermAR</code>	Implementation of <code>EqTerm</code> that represents one or more lags of a (Vector) AR model.
<code>EqTermDefault</code>	Default implementation of <code>EqTerm</code> .
<code>EqTermEC</code>	A special equation term for the error correction term in a VEC model.
<code>EqTermLHS</code>	Equation term to be used for the left hand side of an equation.
<code>SubMatModel</code>	Table column model for the <code>CoeffTable</code> .
<code>VarNamesVector</code>	Displays variable names with or without a time index to indicate a certain lag, e.g. <code>invest(t-1)</code> .

Table 3.39: Elements of Equation system

3.28.3 Architecture Background

The Equation system uses composition to build more complex equation terms from basic default terms. One of the technical problems to solve was to provide a convenient layout for equations with potentially many terms. They have to wrap properly on screen if they get too large to be displayed. The main work for this has been done already by Benkwitz (2002). Layout management does not depend on the concrete implementation classes, but uses the methods defined in the abstract class `EqTerm`. Therefore, it would not need to be changed if new terms are added in the future. One could imagine, for example, having as part of an equation a term defining some function that is common in a certain problem domain.

Using the Equation system can greatly help users to understand the underlying model very quickly without the need to check the help system first. More importantly, it can simplify otherwise very complex user input significantly. For example, the specification of parameter restrictions in models with many variables can be very tedious, especially if there are many restrictions to set. Other programs use text based mechanisms for this, which is also more troublesome to set up from a programmers point of view.

Like the Data Table system, this subsystem also works closely together with the Type System and the Symbol Management. The variable names and coefficients have to be specified via the names of the corresponding symbols in a symbol table of a given scope. It is also possible to specify symbols holding subset restrictions that have been set for coefficients. The information contained in these matrices can be used by the renderer to display the underlying coefficient differently, for example with a symbol '---'. It can also be used to prevent the mouse listener from changing that coefficient. Various possibilities are possible for different modelling situations.

3.28.4 Usage Example

Using the equation system is most efficient with a visual builder tool, although its use is not required. The basic steps are:

1. Use an `EqPanel` as a container for equation terms.
2. Add all needed equation terms in the wanted order. The left hand side term is always displayed first.
3. Adjust the symbol names and possibly other options for the used equation terms. Consult the API documentation for a complete description of all properties.

The code which might be generated by a builder tool could look like the following. It is part of a class that holds an instance of `EqPanel` with a simple equation that is used for specifying restrictions.

```
private EqTermAR endogenousLagged = null;
private EqTermLHS lhs = null;
private EqPanel equationPanel = null;

// setup AR part
private EqTermAR getEndogenousLagged() {
    if (endogenousLagged == null) {
        endogenousLagged = new EqTermAR();
        endogenousLagged.setRendererCoeff(
            JSCCellRendererTypes.SUBSET_01);
        endogenousLagged.setStartLag(1);
        endogenousLagged.setTablePopup(null);
        endogenousLagged.setMouseListenerCoeff(
            JSCMouseListenerTypes.SUBSET_01);
        endogenousLagged.setEditable(false);
        endogenousLagged.setSymbolNameVariables("Ny");
        endogenousLagged.setSymbolNameLags("lags");
        endogenousLagged.setSymbolNameCoeff("Ay");
    }
    return endogenousLagged;
}

// setup left hand side
private EqTermLHS getLHS() {
    if (lhs == null) {
        lhs = new EqTermLHS();
        lhs.setTablePopup(null);
        lhs.setSymbolNameVariables("Ny");
    }
    return lhs;
}
```

```
// setup panel for all terms
private EqPanel getEquationPanel() {
    if (equationPanel == null) {
        equationPanel = new EqPanel();
        equationPanel.add(getLHS());
        equationPanel.add(getEndogenousLagged());
    }
    return equationPanel;
}
```

In the method `getEndogenousLagged`, an AR equation term is initialized. The time index starts with $t - 1$. A special renderer is installed to show symbols instead of number values. A mouse listener manages switching between the valid values 0 and 1 on mouse clicks. Like with data tables, the term is set to be not editable, because editing is done via mouse clicks instead. The popup is removed by setting it to `null`. Finally, the names for the symbols in the global symbol table holding the lags, the coefficients, and the names of the variables are specified. As soon as these symbols change their values, the equation display is updated.

In a similar way, the method `getLHS` sets up the left hand side term. For this component only the variable names have to be specified. In `getEquationPanel` all equation terms are added to an `EqPanel` instance, which should then itself be added to some component.

As always when symbol names are specified, one should notice that it is a good idea to define those names in a single place with instances of `JSCTypeDef` to keep track of each variable definition.

3.28.5 Related View Packets

- Parent: View Packet 20: Components
- Children: none

3.29 View Packet 24: Input/Output

3.29.1 Primary Presentation

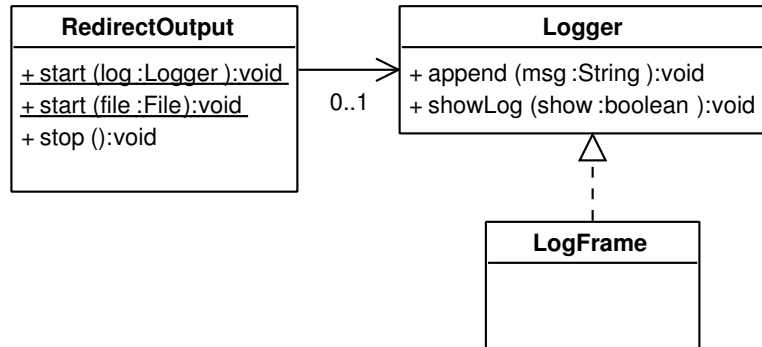


Figure 3.48: Classes of Input/Output system

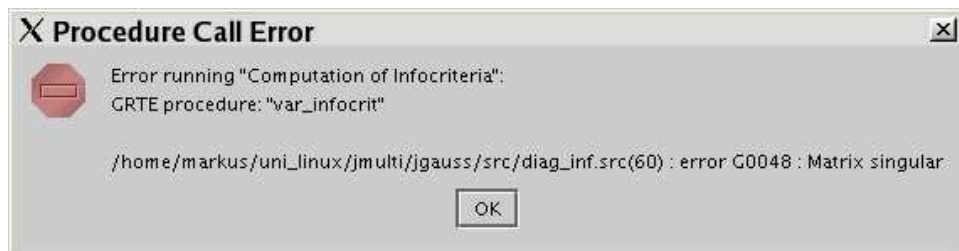
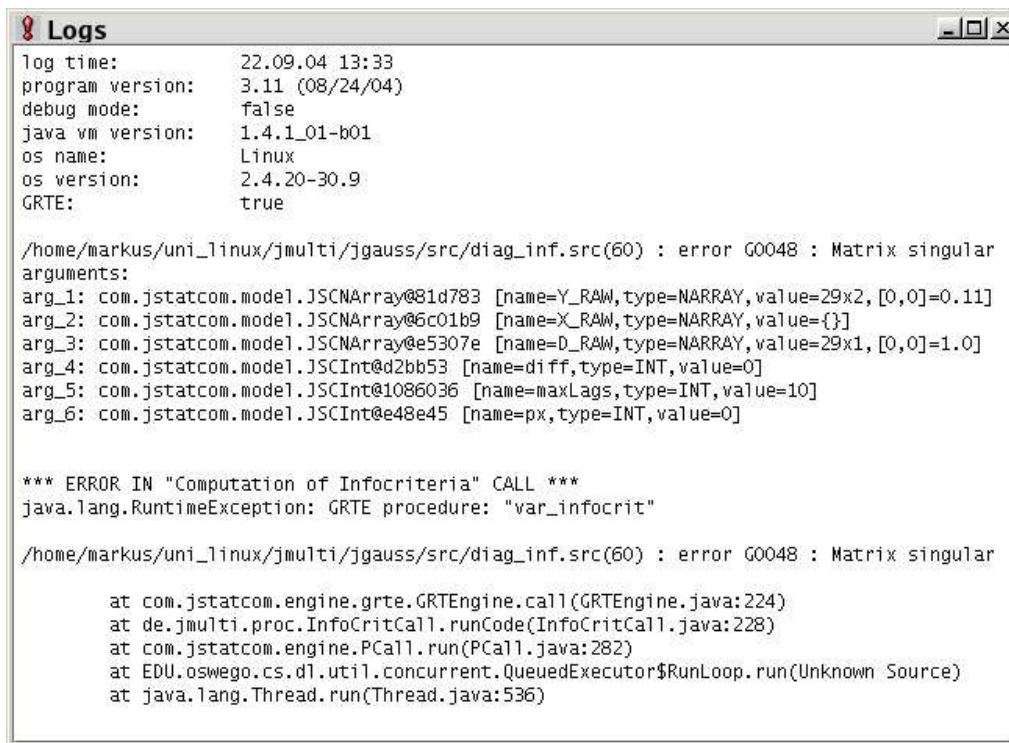


Figure 3.49: Error message presented to the user

The Input/Output system manages tasks related to importing datasets and logging. An important task in applications for data based analysis is to provide a mechanism that deals with detailed error information. Errors might occur due to programming errors in the Java code, but they can also result from wrongly specified input for external procedures. The error message would then be generated by the engine that is used. Typically, errors should result in a message presented to the user. This can be a dialog, as seen in Figure 3.49, or a text output in the result area. However, it often does not make much sense to show all available information to the user, because it is too long and sometimes cryptic to non programmers



```

LogFrame
Log time:          22.09.04 13:33
program version:  3.11 (08/24/04)
debug mode:       false
java vm version:  1.4.1_01-b01
os name:          Linux
os version:       2.4.20-30.9
GRTE:            true

/home/markus/uni_linux/jmulti/jgauss/src/diag_inf.src(60) : error G0048 : Matrix singular
arguments:
arg_1: com.jstatcom.model.JSCNArray@81d783 [name=Y_RAW,type=NARRAY,value=29x2,[0,0]=0.11]
arg_2: com.jstatcom.model.JSCNArray@6c01b9 [name=X_RAW,type=NARRAY,value={}]
arg_3: com.jstatcom.model.JSCNArray@e5307e [name=D_RAW,type=NARRAY,value=29x1,[0,0]=1.0]
arg_4: com.jstatcom.model.JSCInt@d2bb53 [name=diff,type=INT,value=0]
arg_5: com.jstatcom.model.JSCInt@1086036 [name=maxLags,type=INT,value=10]
arg_6: com.jstatcom.model.JSCInt@e48e45 [name=px,type=INT,value=0]

*** ERROR IN "Computation of Infocriteria" CALL ***
java.lang.RuntimeException: GRTE procedure: "var_infocrit"

/home/markus/uni_linux/jmulti/jgauss/src/diag_inf.src(60) : error G0048 : Matrix singular

    at com.jstatcom.engine.grte.GRTEngine.call(GRTEngine.java:224)
    at de.jmulti.proc.InfoCritCall.runCode(InfoCritCall.java:228)
    at com.jstatcom.engine.PCall.run(PCall.java:282)
    at EDU.oswego.cs.d1.util.concurrent.QueuedExecutor$RunLoop.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:536)

```

Figure 3.50: Screenshot of LogFrame with a detailed error message

and might therefore confuse them. On the other hand, most of the time it is the only source of information for the developer to fix an error on the basis of a user response.

JStatCom provides a simple solution to this problem by offering a way to redirect all output that goes to `System.out` and `System.err` also to a logging component. This component must implement the `Logger` interface and can be set at program start via the `RedirectOutput` class. Programmers who want to send messages to the logging system can then use the standard ways of coding print statements, like `System.out.println`. No extra commands are needed. Figure 3.48 shows the related classes. The `Logger` interface has the method `append` which is used to append output and error messages. The method `showLog` should make the component visible. A default implementation of this component is `LogFrame` which is shown in Figure 3.50. As can be seen in the screenshot, it is used to collect normal status messages, as well as detailed error information

that can help developers. The corresponding user message in Figure 3.49 contains much less information. The error message is generated here by the `PCall` class that was involved in the procedure call. It helps to standardize error statements and to trace back the cause of problems quickly. The text field in the logging component uses the `ResultField` class, which allows to easily save the log messages to a file which can be attached to an error report. If the Application subsystem with the default `TopFrame` implementation is used, redirection of output is automatically done at program start. The logging component is then available via a menu. In this case, no extra programming is needed to interact with the logging system, except writing error and status messages to `System.out` or `System.err`. Output redirection can also be used with a log file that should then be specified at program start.

Furthermore, the system contains the class `FileSupport` with various helper methods to save text files, and to create file choosers for selecting input and output files for different tasks. Importing data is a more complex task that involves several classes. It is handled by the Data Import subsystem.

3.29.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.io</code>	
<code>Logger</code>	Simple interface to be implemented by components that process log strings.
<code>FileSupport</code>	Support for convenient file selection.
<code>LogFrame</code>	Frame that can be used as a log area for various kinds of messages.
<code>RedirectOutput</code>	Helper class to redirect stdout and stderr to a log file and/or a logging component.

Subsystems	
Data Import System	Manages importing data from files with different formats. Can be adjusted to load various kinds of data sets.

Table 3.40: Elements of Input/Output system

3.29.3 Architecture Background

During the development of JStatCom and the application JMulTi the importance of a standardized yet easy to use error reporting system became more and more obvious. Often user response is relatively vague and errors are therefore hard to trace back to their origin without detailed information, like for example the stack trace of exceptions that have been thrown. Therefore the logging system was introduced and users should always be encouraged to attach its output when errors are reported. Experience has shown that this discloses the source of errors almost immediately in most cases, thus reducing the time needed for reproducing problems, which is often the most time consuming part of fixing errors.

However, the logging system is only the way of collecting those messages easily. It has to be supported by programmers in such a way, that enough information is provided to be useful for debugging. This means, for example that exceptions that might occur in the program should never be just caught and ignored. By printing the stack trace, they will appear in the log area and can reveal important information. It also means that when exceptions are created and thrown, they should always contain relevant information about the source of the error. An example from the `UMatrix` class is the following method that can be used to create an identity matrix of a certain dimension:

```
public static double[][] eye(int dim) {
    if (dim < 0)
        throw new IllegalArgumentException(
            "Dimension argument " + dim + " < 0.");
}
```

```
double[][] d = new double[dim][dim];
for (int i = 0; i < dim; i++)
    d[i][i] = 1;
return d; }
```

If, for some reason, a negative dimension argument is given, an exception is thrown. The message will include the wrong argument and why the exception was raised. If this message is caught somewhere, the handler should just print the stack trace of the raised exception which would then also appear in the log area. The information given there about the wrong argument, together with the stack trace that reveals information from which method `eye` was called, might already be sufficient to know the source of the error.

One might ask, why this somewhat ad hoc logging system is used instead of the existing logging API provided by the standard Java library since version 1.4. The reason is just that the benefits of using that API are not quite clear in the current problem domain. Experience has shown that the use of well prepared error messages together with stack traces of exceptions is appropriate for the current purpose. More importantly, the use of the logging API would require extra code to produce logging messages. As already mentioned, the current system does not require any extra code, except the usual `System.out.println` and `System.err.println` commands that programmers use anyway. Other advantages of the logging API, like fine grained logging levels, more detailed time information, and a higher flexibility of specifying handlers, are not really needed at the moment. However, programmers can of course use the logging API together with the current system without any problems. Generated log messages would by default also appear in the `LogFrame`, unless special handlers are used.

3.29.4 Related View Packets

- Parent: View Packet 1: `JStatCom`
- Children:
 - View Packet 25: `Data Import System`

3.30 View Packet 25: Data Import System

3.30.1 Primary Presentation

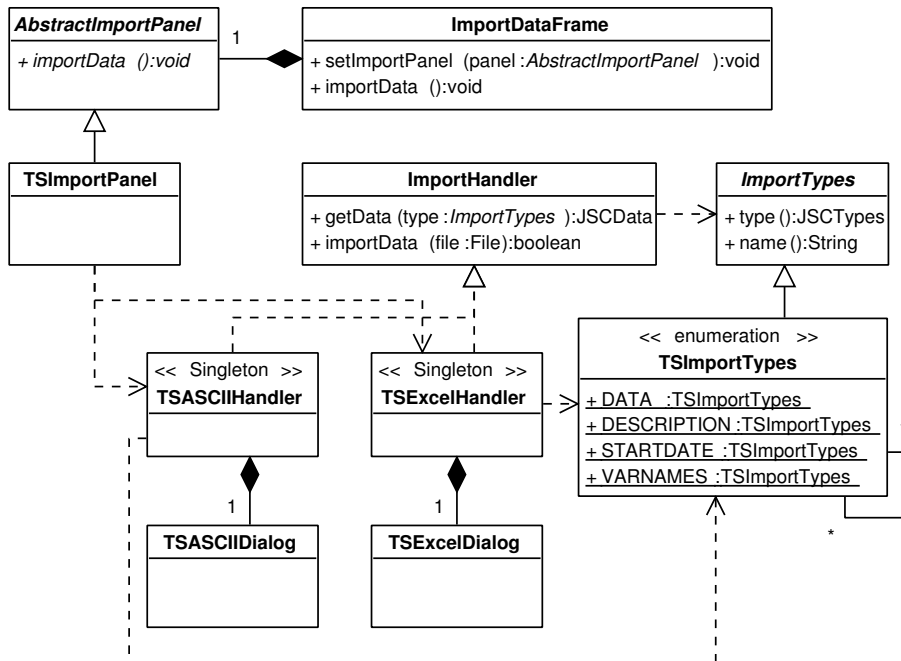


Figure 3.51: Classes of Data Import system

The Data Import system is typically not used by programmers directly, except if they plan to extend it with their own dataset definition. However, this presentation should show possible extension points.

An important part of any data based analysis system is to convert data being stored in files of some format to the data representation that is used internally by the software. This process involves parsing data files and assembling the resulting output in a way that is appropriate for the given problem domain. For time

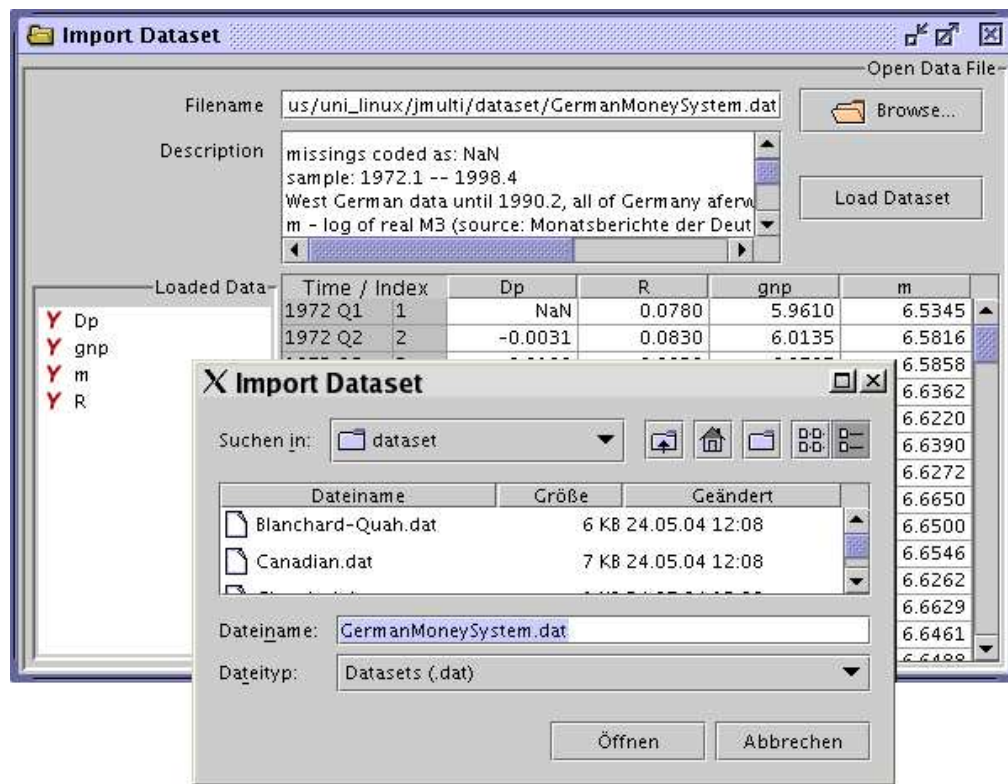


Figure 3.52: Screenshot of ImportDataFrame with a TSIImportPanel

series analysis, this would mean to create a set of time series and add them to the TSHolder from the Time Series system. However, one could also imagine other ways of assembling the output for different problem domains. Therefore, the Data Import system provides an abstract layer which does not make any assumptions about the structure of the underlying dataset nor the format in which the data is stored. It only assumes that all elements of a dataset can be represented as instances of JSCData. Concrete implementations will then define the specific details that are needed to handle data of a certain type.

Figure 3.51 shows the classes of that system. The class ImportDataFrame is just an internal frame that should hold the user interface components needed to import data of some type. How that GUI is designed is not specified in this class, but in an instance of AbstractImportPanel instead. This abstract class must be subclassed to provide the required functionality. The import data frame

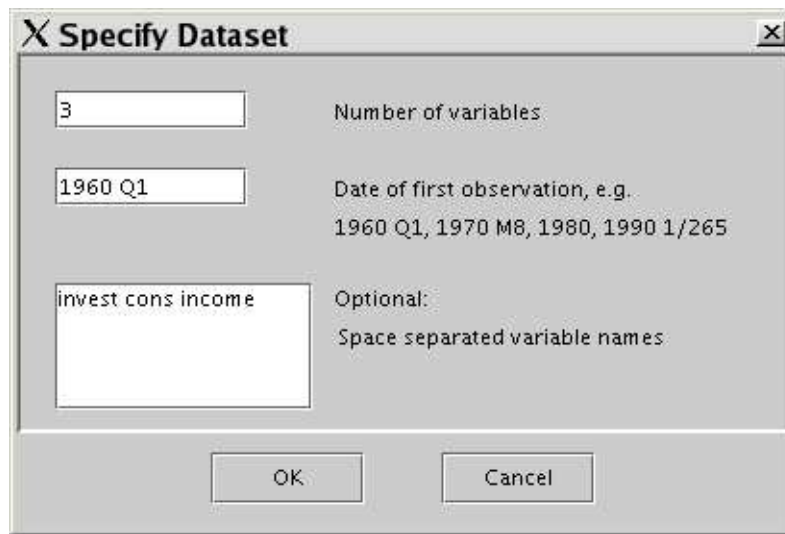


Figure 3.53: Screenshot of TSASCIIDialog

component can be configured with the method `setImportPanel` to tell it which panel implementation to use. This defines the abstract layer for the GUI part of importing data. Figure 3.52 shows a screenshot of it with a panel for importing time series data.

The actual logic for parsing files and assembling datasets is also specified in an abstract layer. Each file handler should implement the interface `ImportHandler`. Parsing data from a file would then be invoked via `importData`, whereas the elements of the assembled dataset can then be accessed via `getData`. Each element of a dataset is identified via an instance of `ImportTypes`.

The two abstract layers are sufficiently flexible to handle any file format and any dataset structure. However, they allow to use that system via some very simple interface methods. All that is needed to invoke a specific import mechanism is to set the panel to use with `ImportDataFrame` and to invoke the `importData` method of that frame. Actually, the Application subsystem does exactly this in the `TopFrame` component. Data import is invoked via a menu item. The default import panel is that for time series analysis, but it could be replaced as well.

For time series analysis, this system has been implemented to assemble time series data from ASCII and Excel files. The two implementations of the `ImportHandler` interface are `TSASCIHandler` and `TSExcelHandler`. They are invoked from `TSImportPanel` according to the used file name suffix, which provides the GUI for reading in time series data. Both handlers parse data files differently, and they can also ask the user for additional information via the dialogs `TSASCIIDialog`, which is shown in Figure 3.53, and `TSExcelDialog`. Although the reading of files in different formats involves completely different techniques, the data for time series analysis must always contain the same types of information. Those are declared in the enumeration `TSImportTypes`, which just defines all elements of a valid dataset. One could easily imagine to add more handlers to parse also other file formats, but the assembled results must always contain the elements laid out in this type definition. This way, they can be accessed from the `TSImportPanel` which creates instances of `TS` and adds them to `TSHolder`.

3.30.2 Element Catalog

Class Name	Responsibility
<code>com.jstatcom.io</code>	
<code>AbstractImportPanel</code>	Abstract panel for general import data functionality.
<code>ImportHandler</code>	Interface to be implemented by classes that initiate parsing of data files and the creation of datasets.
<code>ImportDataFrame</code>	The frame containing the import data functionality.
<code>ImportTypes</code>	Enumeration of constants that define different types of elements that belong to a certain dataset.
<code>TSASCIIDialog</code>	A dialog for gathering user information if an ASCII file is read in.

TSASCIHandler	Import handler for ASCII data files that either contain only the data points or that are in the .dat format.
TSExcelDialog	A dialog for gathering user information if an Excel file is read in.
TSExcelHandler	Import handler for Excel data files.
TSImportPanel	Panel for importing and displaying data files for time series analysis.
TSImportTypes	Enumeration defining the elements of an imported dataset to be used for time series analysis.

Table 3.41: Elements of Data Import system

3.30.3 Architecture Background

The design of the Data Import system was driven by two obvious extension points. First, it is likely that new handlers are being defined to parse different file formats to some dataset. Therefore it seemed reasonable to standardize the general layout of those handlers with an interface `ImportHandler`. The class that invokes the handlers, usually an `AbstractImportPanel` subclass, can then easily switch between different implementations according to the file format that is selected. The invoked methods are always the same, and therefore the calling method does not need to change when the underlying parser changes. This is a typical application of the *Strategy* pattern. The currently implemented strategies are `TSASCIHandler` and `TSExcelHandler`.

The other extension point is that there might also be the need to create a completely new dataset definition that is not compatible with the already implemented time series specification. For example, one could imagine to have an extended specification for cross sectional data, where each series is indexed via time and via a label defining the section it belongs to. This could then be used in a special selection mechanism, especially if a large number of sections is being considered.

3.30.4 Usage Example

To summarize, creating a new parser for an existing dataset specification requires to create another implementation of `ImporterHandler` and to tell the respective import panel when to use it. This can either be done via a user dialog, or via interpreting the file name suffix of the selected data file. The latter might be ambiguous though.

Creating a new dataset specification usually involves subclassing `AbstractImporterPanel` for the GUI, as well as the creation of at least one handler realization. The elements of a dataset should be declared in an enumeration that subclasses `ImportTypes`. This way, datasets get a clear structure and are easy to understand.

A small code snippet demonstrates the usage of the time series import handlers. Originally, this code is part of the class `TSImporterPanel`. It shows the mechanism of parsing different formats and retrieving the assembled information afterwards:

```
ImporterHandler handler = null;
String fName = "datafile.dat";

// select handler according to file name suffix
if (fName.toLowerCase().endsWith("xls"))
    handler = TSExcelHandler.getInstance();
else
    handler = TSASCIHandler.getInstance();
```

```
// import data
if (!handler.importData(file))
    return;

// get imported elements via types
JSCNArray data = (JSCNArray) handler.getData(
    TSImportTypes.DATA);
JSCDate startDate = (JSCDate) handler.getData(
    TSImportTypes.STARTDATE);
JSCSArray varNames = (JSCSArray) handler.getData(
    TSImportTypes.VARNAMES);
JSCString description = (JSCString) handler.getData(
    TSImportTypes.DESCRPTION);
...
```

The example imports data from a file `datafile.dat`. First the appropriate handler is selected and the singleton instance is referenced. In this case, the `TSASCIHandler` is selected. Via `importData` the parsing is invoked. Finally, all elements of the loaded dataset are retrieved from the handler as instances of `JSCData`. The flexibility of the Type System helps to represent each part with an appropriate type. What is missing here is how instances of time series are created and how they are further processed.

3.30.5 Related View Packets

- Parent: View Packet 24: Input/Output
- Children: none

3.31 Concluding Remarks about JStatCom

One of the main goals of creating the framework `JStatCom` was to provide a solution to support the creation of GUI applications for data based analysis that is applicable to a wide range of usage scenarios and that leaves always room for extensions. The design was done in the hope that it can reduce the need for ad hoc

solutions for tasks that occur in literally any application for data based analysis. The description of potential extension points should have pointed interested developers in the right directions.

Another very important goal was to reduce the initial effort to create running applications with the help of JStatCom. GUI developers should not be misled by the fact that almost all systems can be changed and customized. A lot of effort has been put in providing reasonable default values and to make it possible to create a simple module in a few steps. However, as programming experience grows and projects get larger, the framework should still support even challenging development tasks.

The current state of JStatCom can be considered as a core package where all packages are required to let it run. But there are still a few things that have to be added to support development of really full featured and professional applications. First, this is project management, which should allow users to store the state of a model in a file and retrieve it easily from there. Of course, this could be implemented independently of JStatCom in applications using it. But considering that this is not a trivial task, it would be nice to have it as another system on the framework level. This would feature a common XML project file format which could then be adjusted to store specific settings for applications based on JStatCom. The details for each model must then be specified for each module, but it helps a lot if there is already a general infrastructure. Some design decisions that have been made had this vision of a project management system already in mind. For example, it is possible to store all JSCData instances in XML files, even whole symbol tables can easily be serialized and deserialized. This mechanism together with the Symbol Management system will help to make it relatively easy to define project settings and to restore them from files if symbol tables are used to represent the state of a model.

Another very desirable extension would be to have a feature-rich Java based graphics engine. This is, because creating graphics via external engines is often somewhat troublesome and does not result in a consistent user experience.

Developing such a graphics engine is beyond the scope and also against the spirit of this project, but there are already quite powerful toolkits that can be reused. A good candidate is the already mentioned *jfreechart* library which appears to have a very active developer community and is constantly being improved.

Finally, it is always possible to use the extension points of the framework to add new functionality, like for example new data types, more parsers for data import, other engine implementations, or special GUI components. It is argued that JStatCom is a very flexible solution to design GUIs for algorithms that can be used to analyse data. However, the drawback against other solutions is that some amount of Java programming is needed. This might be considered being too complicated, especially by researchers who want to do the GUI development themselves. Often they prefer to work with a programming language they already know, for example Matlab. However, it is hoped that JStatCom is used in projects where more complex GUIs are required and where the offered flexibility is needed. Its big advantage is that it is based on a standard technology that is supported by various tools and rich documentation. Furthermore, Java is being taught in many introductory courses on programming and informatics and it should not be too difficult to find people capable of filling the role of the GUI developer.

To ensure that JStatCom is being further developed, maintained, and supported, a possible solution would be to turn it into an open-source project. This can be used to attract other developers to join the development team. However, this must also be supported by tools because joint development poses coordination costs. The infrastructure for hosting such projects is available for free at *sourceforge.net*. This site offers various features that are essential if many developers need to work jointly, for example version control, release management, bug tracking, discussion forums, etc..

The author has already gained some experience with hosting and maintaining an open-source project with the software JHelpDev.⁷ Because JStatCom might be of general interest to the Java community, a better alternative to *sourceforge.net*

⁷JHelpDev is hosted at sourceforge under *jhelpdev.sourceforge.net*. The project is led by the author and by now there are two registered developers who have contributed to the integration of a number of features, for example a smart automatic table of contents generation algorithm.

would be to host the project at *java.net*. This platform offers similar services than sourceforge but is more specific to Java. Therefore it might be easier to promote JStatCom there and to find fellow developers. This might also be a solution that allows the author to step back from this project when others have taken over. However, first it needs to be migrated there and published to the community.

So far the response to presentations of JStatCom was very positive, especially by researchers who have experience with programming GUIs for their modules. There is definitely a request for supporting this task. This can also be seen from the fact that recently a somewhat similar project has been started at the mentioned *java.net* site which is called *JDesktop Network Components (JDNC)*.⁸ In the project description it is written:

The goal of the JDesktop Network Components (JDNC) project is to significantly reduce the effort and expertise required to build rich, data-centric, Java desktop clients for J2EE-based network services.

From this goal it can be seen that there is a need for frameworks that support the creation of GUIs for applications that need to represent data. As compared to JDNC, JStatCom has a stronger focus on writing user interfaces for algorithms. However, there are overlapping goals and JStatCom can certainly benefit from components provided by JDNC. Because both projects are written in Java it would be straightforward to integrate JDNC classes.

Because JStatCom was designed to be usable not only for time series analysis and econometrics, it should be an attractive choice also for developers in other problem domains, for example in engineering. For this it will probably be necessary to add support for new file formats to import data of other types than time series. It is argued that JStatCom is a good starting point for programming analysis tools quickly. Its data management and GUI capabilities should be general enough to satisfy the needs of developers in various fields.

⁸The JDNC project homepage is jdnc.dev.java.net.

Chapter 4

JMulTi - A Reference Application of the Framework

4.1 Introduction

The framework JStatCom that has been described in the previous chapter was developed from the experiences gained while creating the software JMulTi. Originally, that software was designed as a convenient GUI to complex and difficult to use econometric procedures written in Gauss, that were not available in other packages. Because this concept has proved to be quite fruitful, JMulTi has evolved to a comprehensive modelling environment for multiple time series analysis.

As the software was getting more and more complex, it also became clear that it could only be maintained on a long term basis, if sound software engineering principles would be followed. It was also obvious that the software concept would certainly be a useful approach for many related applications as well. This has motivated the development of the framework JStatCom, which was already initiated by Benkwitz (2002).

The current chapter describes the application JMulTi in some detail from a developers point of view, and, to a lesser extent, from a user's viewpoint. Because so far this software is the only program that uses JStatCom, not counting showcase examples, the discussion of its design might help developers to layout and create their own modules. The benefit of using a framework is also that applications

based on it have a very similar structure. Therefore, the design of JMulTi modules can almost serve as a blueprint for other modules as well. Furthermore, the subsystems that have actively been used will be referenced throughout this chapter. After all, the following descriptions should show what can be accomplished with the framework. The author has initiated the development of all modules currently in JMulTi, except *VAR* and *VEC*. However, also these parts of the program were enhanced with various features and have been almost completely rewritten. Details about the underlying econometric procedures will not be in the focus of this text. For this, the help system of JMulTi can be consulted, as well as the comprehensive reference Lütkepohl and Krätzig (2004).

It should be mentioned that JMulTi has meanwhile become a software that is actively used for empirical research and teaching. Due to response by users it was possible to improve the program over time and to fix various errors. Particularly instructive was a workshop organized in March 2003 in Florence which gave a profound feedback to the author, thus pointing to many enhancements.

4.2 General Setup

The general usage of JMulTi is pretty much predetermined by the layout of the framework JStatCom. The software makes use of the Application system (Section 3.26). Functionality that belongs together is summarized in modules. Each module appears in its own internal frame and can be used separately. One can identify two levels of interaction:

- *General functions* - Those tasks are implemented on the framework level and are shared by all modules. They include the symbol control system (Section 3.12), the time series calculator (Section 3.24), the data import system (Section 3.30), the log frame (Section 3.29), as well as the help system. In future versions this will also include project management and services related to graphics.

- *Module specific functions* - Each module provides a GUI to a certain set of procedures that follow a common theme. The modules make use of the services provided by the framework, for example, they use the data that has been imported. Modules are otherwise largely independent of each other.

This basic layout of JMulTi is similar to the design of the Oxmetrics family of econometric software. Like JStatCom, the GiveWin framework provides a general infrastructure that can be extended with specialized modules. It is an interesting observation that this similarity has appeared, although the two systems have been developed independently. Therefore, it is identified as a useful pattern for data based analysis. However, JStatCom is a more general development framework than GiveWin/OxPack, because it allows more freedom to design the individual modules and to operate with different engines.

It is also worth thinking about the term *Reusability* in the context of the software JMulTi. For JStatCom, reuse was one of the main design goals for all classes and subsystems. But applications that are based on JStatCom can provide reusable components themselves. However, software reuse is typically not one of the main aspects anymore, because it requires considerably more effort in terms of documentation and class design. Therefore, reusability is often a mere side effect of application development. What is more important, is to avoid repeating code within the application. This often requires internal reuse of classes which is much easier to achieve than external reuse, because the usage scenarios are more clearly defined. External reuse means that parts of JMulTi could programmatically be invoked from within another software. For example, one could imagine that the VAR estimation procedure would be called from another Java application, thus making this routine available in other packages as well.

JMulTi does reuse classes internally for tasks that occur in similar or only slightly modified ways in several parts of the program. For example, several tests for residual analysis are used a couple of times across different modules. It would certainly be bad design if those procedure calls were always implemented again. Therefore the PCall system (Section 3.19) was used to encapsulate those procedure calls in separate classes that can be used from different components easily.

4.3 Modules of JMulTi

Having described the very general design of JMulTi which is imposed by the underlying framework, this section gives an overview of the modules that are actually being implemented.

First, it should be mentioned that JMulTi uses the GRTE (Section 3.15) or the Gauss engine (Section 3.14). All modules rely on the external resources that have been described in the respective parts of the architecture documentation. The default mode for JMulTi is to work with the GRTE. The Gauss engine is only used for development and debugging purposes. It is possible to switch between those two engines via the command line option `-DGRTE="true"` or `"false"`.

Table 4.1 presents all modules that have been implemented so far. They will be described in greater detail in the remaining sections of this chapter. The user of JMulTi is expected to start a time series analysis by applying the module *Initial Analysis* to investigate basic properties of the data and to decide on stationarity of the single series, as well as to test possible cointegration relations. Afterwards, the user might choose one of the other analysis modules to specify and estimate certain models.

The table also lists the implementation units for each module. All classes have the common prefix `de.jmulti` and are grouped in packages with descriptive names. It can be seen that `proc` and `tools` are used by all modules. Those packages contain classes and components that are reused internally. The `proc` package holds all `PCall` implementations for each procedure call that is being made in JMulTi.

Analysis Module Implementation Units (de.jmulti)	Description
Initial initanal tools proc	Entry point for time series modelling. Provides a workbench panel with descriptive statistics, spectrum, autocorrelation, and kernel density analysis. Offers a range of unit root and cointegration tests.
VAR var tools proc	Specification and estimation of full and subset VAR models with impulse response analysis, diagnostic checks, forecasting, and more. Also offers the possibility to estimate SVAR models.
VEC vecm tools proc	Specification and estimation of VEC models with impulse response analysis, diagnostic checks, forecasting, and more. Also offers the possibility to estimate SVEC models.
ARCH arch tools proc	Allows to estimate univariate volatility processes with different error distribution assumptions and ARCH, GARCH or TGARCH specifications. Multivariate MGARCH estimation is possible as well.
STR str tools proc	Specification and estimation of STR models, as well as nonlinearity tests. All parts of the estimated STR model can be plotted.
Nonparametric cafpe tools proc	Allows to specify, estimate, and analyze univariate nonparametric time series models for the conditional mean and the conditional volatility of a stochastic process. Forecasting is possible as well.

Table 4.1: Modules of JMulTi

It would always be possible to add newly defined modules without interfering with the existing ones, given that there are no compatibility issues with the underlying external resources. If the new modules are also making use of the GRTE engine, then all Gauss sources should be compiled into a single .gcg file. This means that there must not be any naming conflicts with the procedure names or global variables.

4.4 How to read this Chapter

The following sections describe each module from different viewpoints. Readers should get an idea of what a module is meant for, and they should also learn how the module was built by making use of the features provided by JStatCom. Like in the previous chapter, it has been found to be useful to stick to a general structure:

1. *Overview*: Describes the general usage context of the module, typically with screenshots.
2. *Implemented Features*: Gives a detailed account on the functionality that is provided by the module. This is helpful from a users point of view to evaluate the software in terms of applicability for her purposes.
3. *Implementation Details*: Describes selected solutions for the implementation of certain features. This section should be helpful for developers, because it might relate to what they are trying to accomplish. It also links to relevant sections of the architecture documentation. In some cases, code examples are used.

4.5 Initial Analysis

4.5.1 Overview

The Initial Analysis consists of tasks that are typical for the beginning of any time series analysis. First, the user should get an idea of the data to be analyzed. This can be done via checking plots, descriptive statistics, autocorrelation functions,

spectrum, and kernel density estimates. Direct dependencies between two series could be investigated with crossplots. Figure 4.1 shows the textual output of the computation of the AC and PAC functions for a selected series.

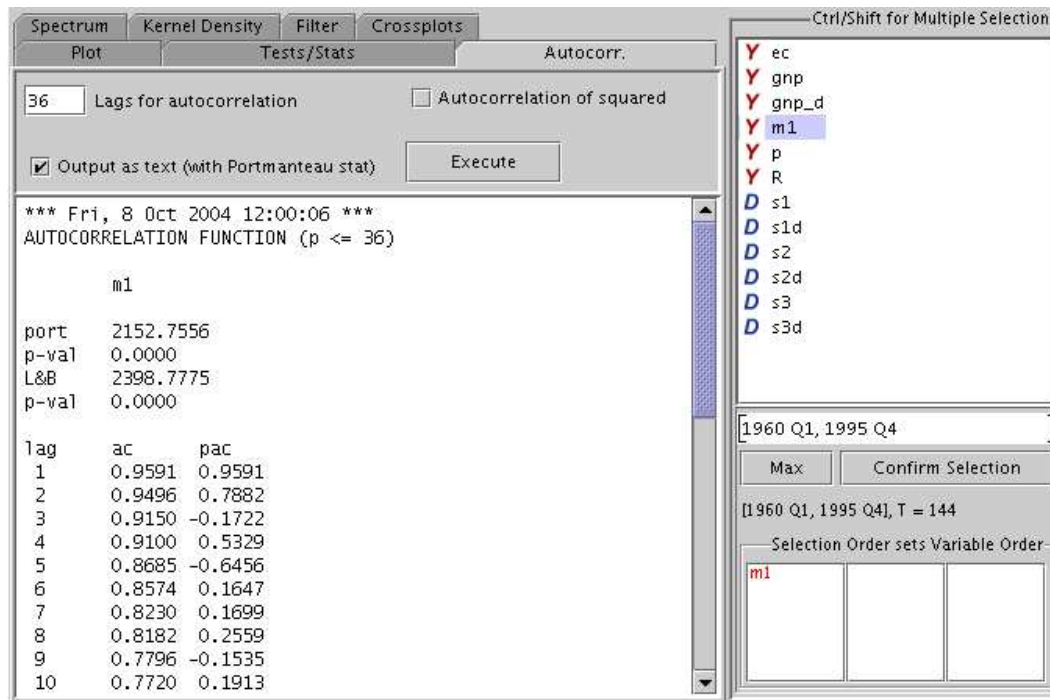


Figure 4.1: Screenshot of workbench with autocorrelation panel

Another important part of the analysis should be to check the stationarity properties of the series used for a model. This can be done via the unit root test panel, which is shown in Figure 4.2. There is a range of test procedures that can be applied, and a selection box allows to switch between panels for different tests.

When there is actually instationarity being discovered, cointegration tests might help to determine, whether a stable long run relationship exists between the series. The number of those relations can be determined as well. The outcome of this test can help to decide which model to use for the further analysis. Figure 4.3 has a screenshot of one of the implemented cointegration tests.

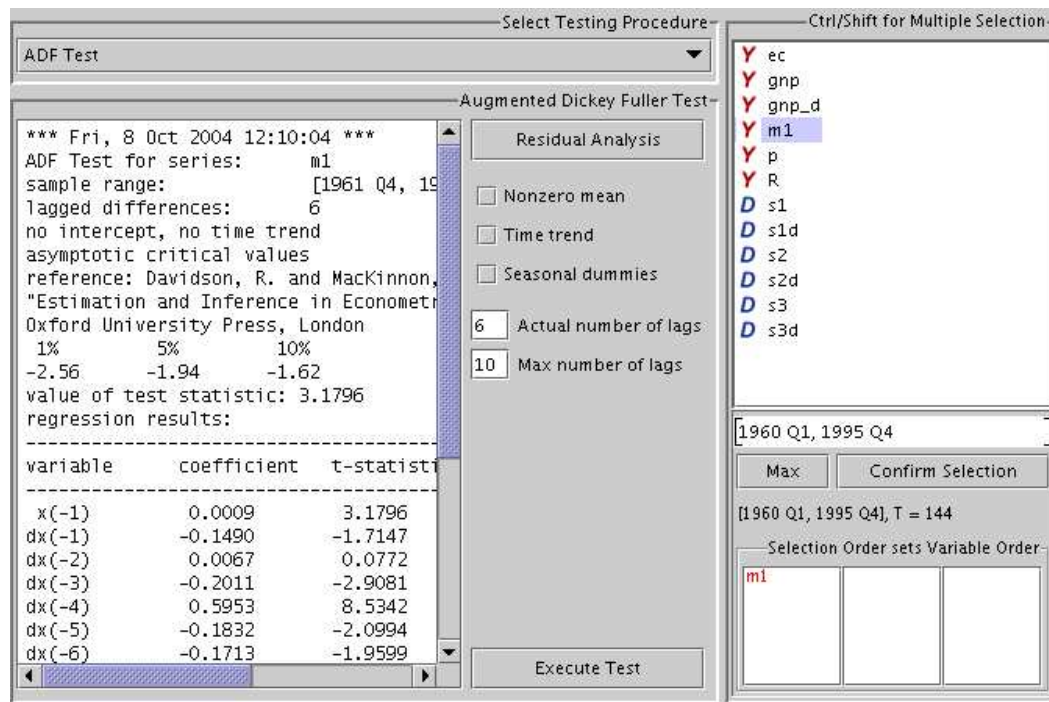


Figure 4.2: Screenshot of ADF unit root test panel

4.5.2 Implemented Features

Workbench

- **Plots** - This panel can be used to configure plots of the selected variables in several ways, including indexed and standardized plots.
- **Descriptive Statistics** - Shows mean, minimum, maximum, standard deviation and variance of the selected series. It also computes the Jarque-Bera tests (Jarque and Bera (1987)) and ARCH-LM tests (Engle (1982)) for each series.
- **Autocorrelation** - Computes the AC and PAC functions of the selected time series up to a maximum lag order. The results can either be plotted or printed.
- **Spectrum** - The ACs of a stationary stochastic process may be summarized

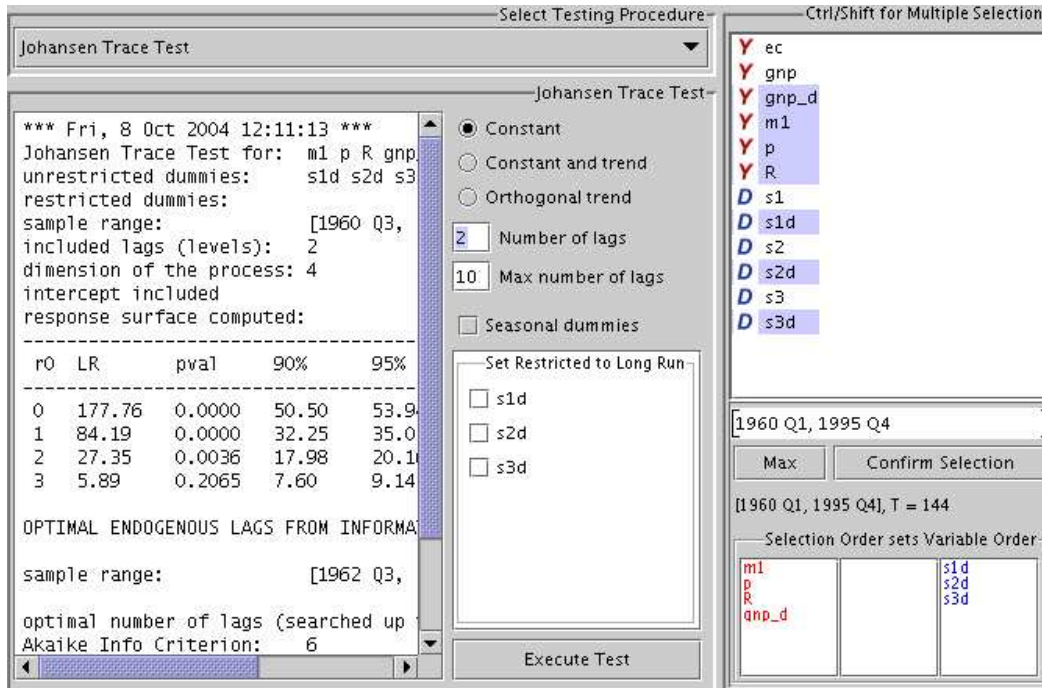


Figure 4.3: Screenshot of Johansen cointegration test panel

compactly in the *spectral density function*. It is defined as

$$f_y(\lambda) = (2\pi)^{-1} \sum_{j=-\infty}^{\infty} \gamma_j e^{-i\lambda j} = (2\pi)^{-1} \left(\gamma_0 + 2 \sum_{j=1}^{\infty} \gamma_j \cos(\lambda j) \right) \quad (4.1)$$

where $i = \sqrt{-1}$ is the imaginary unit, $\lambda \in [-\pi, \pi]$ is the frequency and the γ_j are the autocovariances of y_t . In JMulTi this function can be estimated and either be plotted or printed.

- **Kernel Density Estimation** - With this panel, it is possible to estimate the kernel density function of the selected series with a number of different kernels. The result can either be plotted or printed. Further options are computation of the first derivative of the estimated density, standardization of the selected series, computation of the density for a generated standard normal variable for comparisons, as well as adjusting the range and the bandwidth for the kernel estimator.

- **Crossplots** - Sometimes it is useful to investigate the direct relationship between two variables. Crossplots offer an intuitive graphical tool to look at comovements between two different variables. It may also be helpful to compare the plot with a simple OLS regression line, as well as with a nonparametric estimate.

Unit Root Tests

- **Augmented Dickey-Fuller (ADF)** - A panel for specifying the ADF test (Dickey and Fuller (1979)). It computes the test regression and presents the estimation results, the critical values and the test statistic, as well as the number of lags suggested by the information criteria AIC (Akaike (1973)), HQ (Hannan and Quinn (1979)), SC (Schwarz (1978)), and FPE (Akaike (1969)).
- **HEGY** - Computes the HEGY test to check for seasonal unit roots in quarterly (Hylleberg et al. (1990)) and monthly (Franses (1990)) time series. The panel presents the results of the test regression, critical values and test statistics. As with the ADF test, the lags suggested by the information criteria are given as well.
- **Unit root test with structural break** - Computes a unit root test allowing for a level shift in the mean that follows a certain shift function (Saikkonen and Lütkepohl (2002), Lanne et al. (2002)). It can also be used to find the break point first. The panel shows the results of the test regression, the critical values, and the test statistic. As with the ADF and HEGY tests, the lags suggested by the information criteria are given as well.
- **Residual Analysis** - For the ADF, HEGY, and unit root test with structural break, diagnostic tests for the estimated residuals are provided. They include plotting the residuals, plotting the AC and PAC functions, the Portmanteau test (Ljung and Box (1978)), as well as the Jarque-Bera and ARCH-LM tests.

- **Schmidt-Phillips** - This panel computes a unit root test allowing for the presence of a deterministic linear trend, as proposed by Schmidt and Phillips (1992). The output contains the test statistic, as well as critical values.
- **KPSS** - A test for the null hypothesis of stationarity (Kwiatkowski et al. (1992)). The panel computes the test statistic with either only an intercept, or an intercept and a trend as deterministic part. Critical values are presented as well.

Cointegration Tests

- **Johansen** - Computes the Johansen trace test (Johansen (1995)) to determine the cointegration rank of a set of $I(1)$ variables, see Figure 4.3. It can be executed with three different specifications for the deterministic part:
 - Unrestricted mean term and no linear trend
 - Constant and linear trend
 - Trend orthogonal to cointegration relations

The test panel also allows to restrict the selected deterministic variables to the EC term. Because the lag order of the underlying VAR model must be specified, information criteria can be used to choose the correct number. The critical values as well as the p-values of all Johansen trace tests are obtained by computing the respective response surface according to Doornik (1998) if there are no breaks, or according to Johansen et al. (2000) if there are up to 2 breaks. If the p-value cannot be computed for a given model, then only the test statistic is presented.

- **Saikkonen & Lütkepohl** - Computes the S&L cointegration test (Saikkonen and Lütkepohl (2000, b, c)) which first estimates the deterministic term by a GLS procedure and subtracts it from the original series afterwards. Then a Johansen type test is applied to the adjusted series. Like the Johansen test, it can be executed with three different specifications for the

deterministic part. The lag order may be specified with the help of information criteria. The p-values for the test statistics are generated with a response surface according to Trenkler (2004).

4.5.3 Implementation Details

By looking at the three Figures 4.1, 4.2, and 4.3, a similar pattern for the GUI design of those panels can be found. There is always a selection panel, some controls for input specification, and a text area for showing the generated output of a procedure. This is a common theme for many panels in JMulTi. The framework JStatCom greatly facilitates the creation of such user interfaces, because it provides specialized components for it.

The selection mechanism is always handled by a `TSSel` component, see Section 3.23 for a description. This class is typically used at least once in each of the modules for time series analysis, because variables always need to be selected for a model. For the unit root tests, it is adjusted to allow for the selection of only one endogenous variable, because the underlying models are univariate. This is different for selecting variables for the cointegration tests, because here a VAR system can be specified. But exogenous variables are disabled in this setting, because they are not supported by the test routines.

The text output from the procedures is always presented with an instance of the `ResultField` class, see Section 3.25. It holds the generated output and provides a helpful popup menu for saving, clearing, printing, and adjusting the font size.

Another important component that is visible in all three screenshots is the `NumSelector` textfield, see Section 3.25. It appears whenever users should input numbers and provides input checks that can also be used to validate against some interval. In the Initial Analysis, most number inputs are lags for the underlying regressions. Therefore the number input fields take integer values. A different behaviour of that component is seen in the panel for kernel density estimation, where the bandwidth can be set manually. This field takes a real number up to a precision of 0.1, which is sufficient for that case.

Figure 4.3 also shows a `CheckBoxList`, see Section 3.25, which is used to select boolean properties for a number of objects. Although this could in principle

also be done with a default list, the checkbox list is a more intuitive way of doing this. Furthermore, the selection of the contained items is kept even if elements are being removed or added, which is different from the default behaviour of a `javax.swing.JList`.

To summarize, it should be clear that the components provided by `JStatCom` help to develop GUIs rapidly, because they can bring complex functionality into the application. It should also be mentioned that the Symbol Management system is an integral part of programming with these components, especially with the `TSSe1` class. The workings of it have been described in Section 3.12. Having this system working in the background relieves the GUI developer from the task of thinking about how to exchange variables between the components involved. It suffices to set the names of the variables under which the selection component stores the data in the symbol table. Other components can then easily retrieve the values from there.

Figure 4.4 shows a screenshot of the symbol control frame at runtime with the contents of the global symbol table for the Initial Analysis, as well as the local symbol table for the ADF test. It can be seen that the selected variables are all stored in the global symbol table. For the unit root tests this is `UR_DRANGE`, `UR_ENDDATA`, `UR_ENDNAMES`. These are used by all unit root test panels. Each specific test panel stores its results in a local symbol table. This is not really needed for the program to run, but should bring some transparency to the potential user. This way, for example, the estimated residuals from the ADF regression can be exported. They are stored under the variable `ADF_RESIDS`. One could ask here, why the selection for unit root tests is stored in the global table, because it is not of interest for cointegration tests or the workbench analysis panels. The answer is that it would have been equally possible to split the symbol table hierarchy into separate branches. This was not done, simply because there are only relatively few variables that are actually being shared. However, for more complex GUIs this would be a better solution.

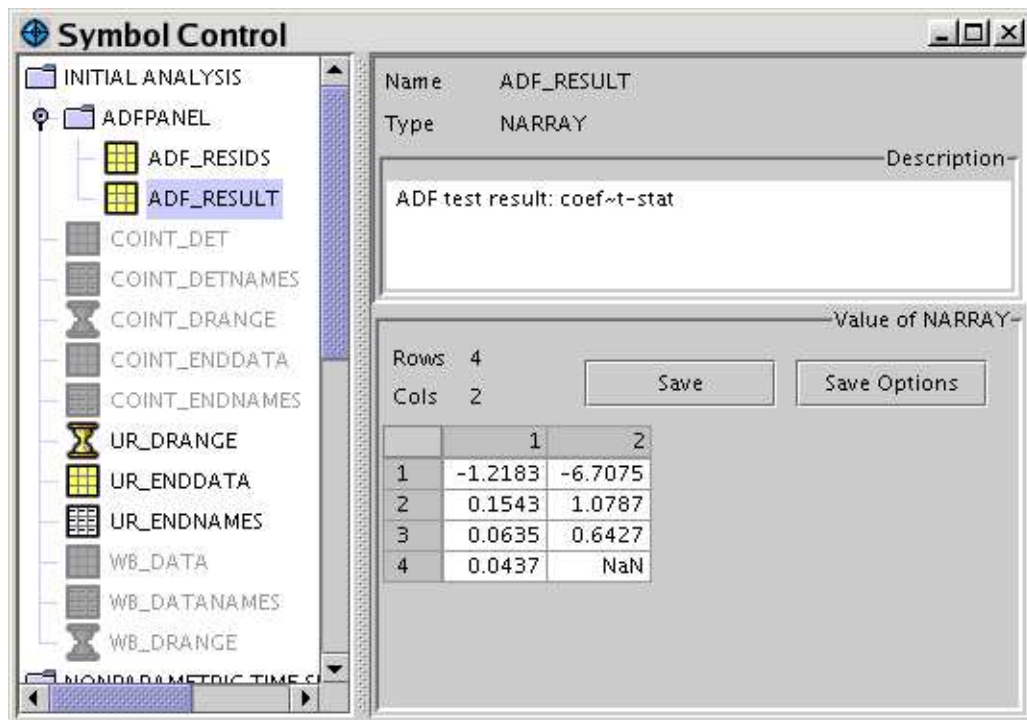


Figure 4.4: Screenshot of Symbol Control for Initial Analysis

Organization of Classes

Figure 4.5 gives an overview of the classes that are used for the Initial Analysis. The top level component of that module is `InitAnalFrame`. It contains three panels that hold the analysis parts. The graphic only reveals more details for the `CointPanel`. It holds a `TSSel` component for selection and a container that itself has the two panels for the cointegration tests. Each of those panels uses a `PCall` subclass to invoke the test procedures. All relations are compositions, which is typical for GUI design. The class organization was done in a way that each class gets a distinct functionality, and that it is not overloaded with too many features. As an extreme case, one could also have put all panels in a single class, but that would not have been good class design, because it would have led to a system that is hard to maintain and extend. However, designing class structures for GUIs is normally very straightforward. The simple proposed rules are:

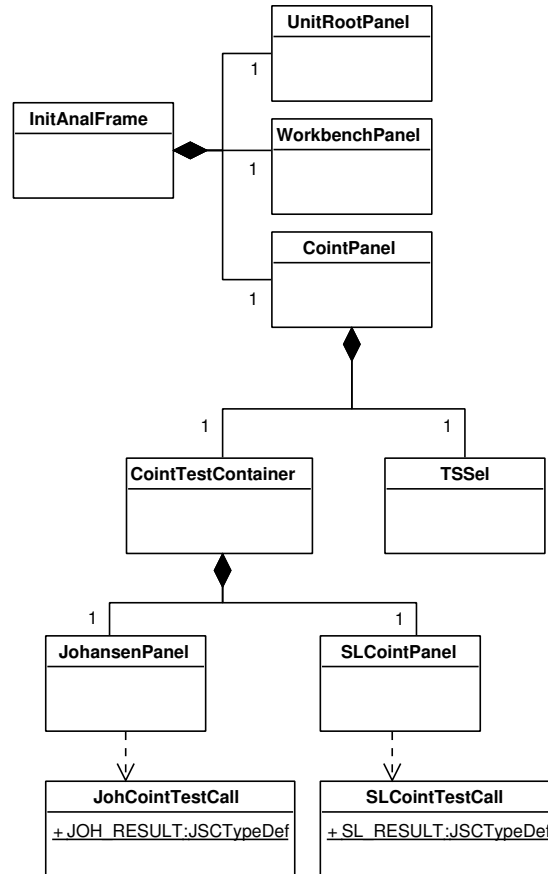


Figure 4.5: Classes for Initial Analysis

- Create a separate class for each GUI that has a specific task. Those tasks can typically easily be identified, for example executing the ADF test, the Johansen test, etc..
- Put the panels together in appropriate container panels. For example, all panels for unit root tests and cointegration tests are put in a container panel that allows to choose which test panel to use.

- Implement the procedure calls by using the Engine and PCall systems, see Sections 3.13 and 3.19. Invoke those calls from the GUI panels for the respective procedures, usually via an *Execute* button.

The last point needs some more explanation. So far, only the GUI components were described. However, the procedures must be invoked somehow from the specification panels. JStatCom supports this via the Engine and PCall systems. Developers should always subclass `PCall` to implement a procedure call for each of the invoked algorithms. This class provides many features that are useful in that context, like using an extra thread to run, a standardized error handling scheme, support for listeners, interaction with a status display, and more. By creating an extra class for each procedure call, it is possible to test those calls automatically without the need to start the GUI component first. Although this seems to be a mere technicality, it can greatly help to deliver quality modules and to improve the code. A second advantage is that calling logic and output formatting are separated from the GUI components and can therefore separately be maintained. Furthermore, procedure calls that are in extra classes can also be called from different parts of the GUI. This is what is meant by internal reuse. In the Initial Analysis, the diagnostic checks for the residuals always use the same procedure calls, although they are invoked from three different test panels, namely ADF, HEGY and UR test with structural break. Those calls are also used by other modules when diagnostic checks for univariate models are carried out. Therefore all caller classes are organized in the package `de.jmulti.proc`, whereas the module specific GUI components go into distinct packages, like `de.jmulti.initanal`.

The two `PCall` classes from Figure 4.5 also have publicly accessible variable definitions `SL_RESULT` and `JOH_RESULT`. Those fields are instances of `JSCTypeDef` and define the name, type, and description under which the test results are stored in the symbol table. Which symbol table is actually used must be set by the invoking GUI component. In the current case, this is the local symbol table, because the test results are not processed by any other panels, but should be shown in the Symbol Control for transparency. In the same way, all procedure call objects in the Initial Analysis define how the computed results are stored. This should also be the preferred way for any new caller class to specify variables holding the

results. However, the only exception is when the number of computed data objects is large. In that case it will pay off to use a separate class with the variable definitions.

Some components are also of interest to other modules, for example the panels for computing the spectrum and the kernel density estimation. They are applied in all other modules for the residual analysis. For this reason, those components can be reused as a whole, meaning GUI panel and procedure call class. All components that are to be reused internally are summarized in the package `de.jmulti.tools`. Component reuse is very effective, because it allows to incorporate complex functionality quickly. However, the components must be flexible enough to be adjusted to slightly different modelling situations.

4.6 VAR Analysis

4.6.1 Overview

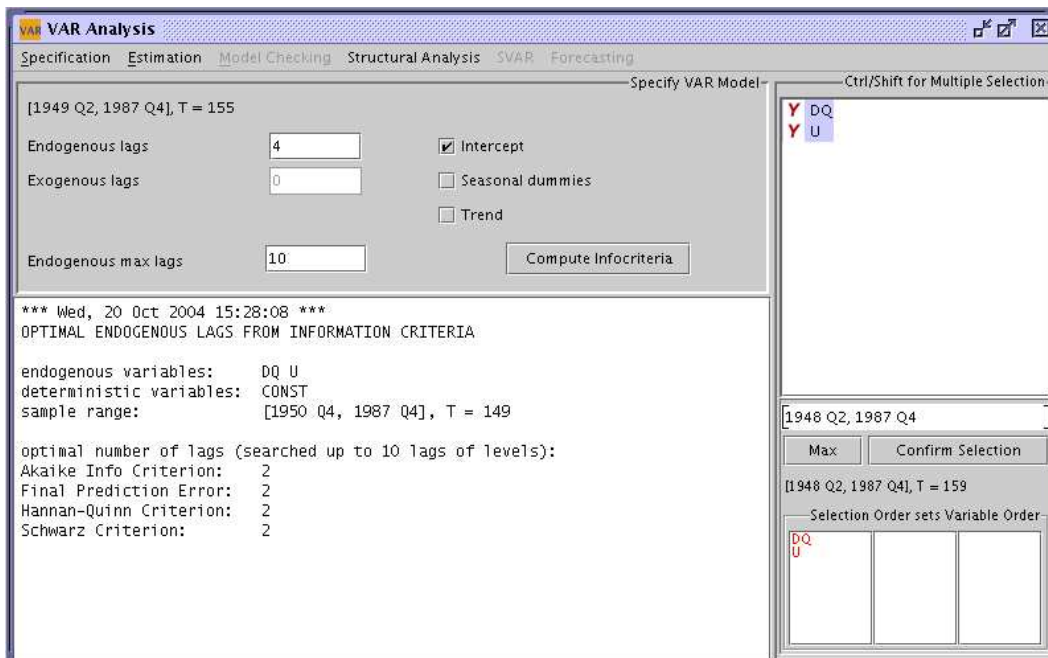


Figure 4.6: Screenshot of specification panel for the VAR analysis

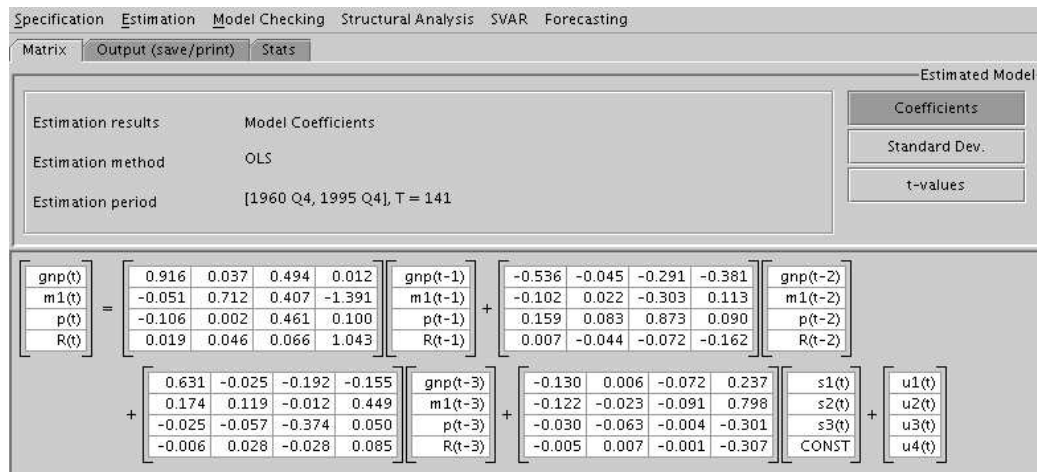


Figure 4.7: Screenshot of estimation panel for the VAR analysis

Finite order VAR models can be specified, estimated, analyzed and used for forecasting in JMulTi. The module allows to analyse VAR models of the form

$$y_t = A_1 y_{t-1} + \dots + A_p y_{t-p} + B_0 x_t + \dots + B_q x_{t-q} + C D_t + u_t, \quad (4.2)$$

where $y_t = (y_{1t}, \dots, y_{Kt})'$ is a vector of K observable endogenous variables, $x_t = (x_{1t}, \dots, x_{Mt})'$ is a vector of M observable exogenous or unmodelled variables, D_t contains all deterministic variables, and u_t is a K -dimensional unobservable zero mean white noise process. Deterministic variables may contain a constant, a linear trend, seasonal dummy variables, as well as user specified dummy variables. All basic properties of the model, like variables, sample range, lags, can be selected in the specification panel, see Figure 4.6.

The A_i , B_j and C are parameter matrices of which the elements are estimated. JMulTi presents the estimation output in an intuitive form via a matrix display that resembles the mathematical notation given in Equation (4.2), see Figure 4.7.

To help to determine the lag order p of the VAR model, model selection criteria can be applied. They are also available via the specification panel (Figure 4.6). Furthermore, various restrictions can be imposed on the parameter matrices. In particular, zero restrictions can be set via the *Subset Specification* panel in JMulTi, which is presented in Figure 4.8. It is possible to set subset restrictions

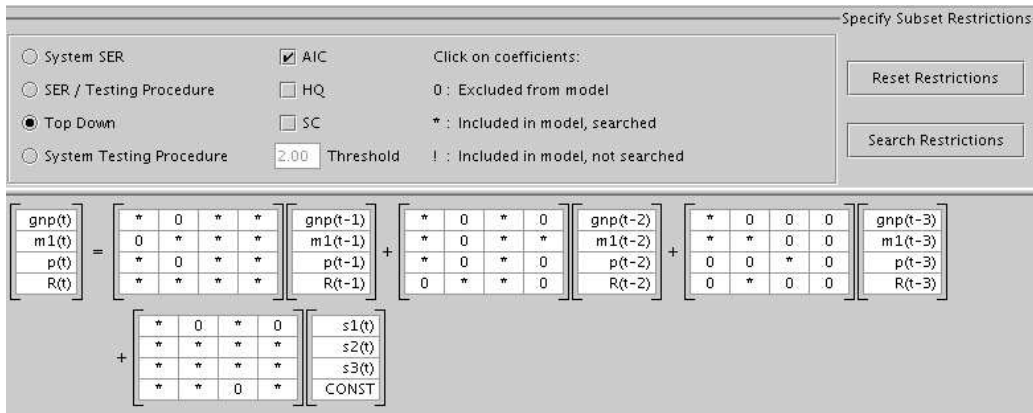


Figure 4.8: Screenshot of manual/automatic subset specification for the VAR analysis

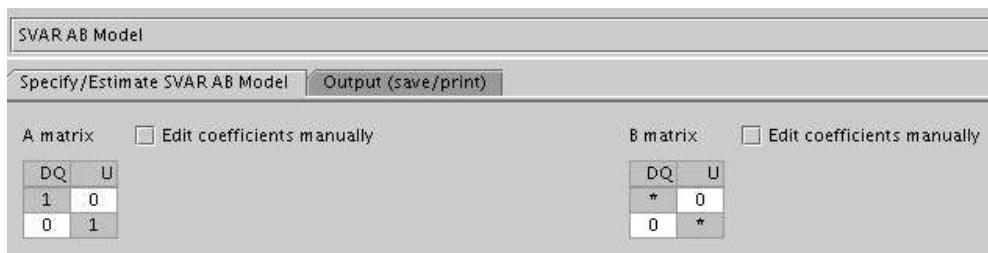


Figure 4.9: Screenshot of SVAR model estimation

manually via mouse clicks over the respective elements of the parameter matrices, but there is also the option to apply a range of model reduction strategies to find zero restrictions automatically according to a selected criterion (Brügemann and Lütkepohl (2001)).

It should be noted that the general VAR model in Equation (4.2) nests the univariate AR model with just a single endogenous variable. Thus the VAR analysis module can also be used for AR models.

The VAR module provides GUI panels for all modelling steps that can be accessed via the menubar of the module frame, as can be seen at the top of Figure 4.6. The general idea of the underlying methodology is that model building is a stepwise process that can partly be automated, but that is steered by the experienced user. Therefore, JMulTi supports the user in choosing the appropriate

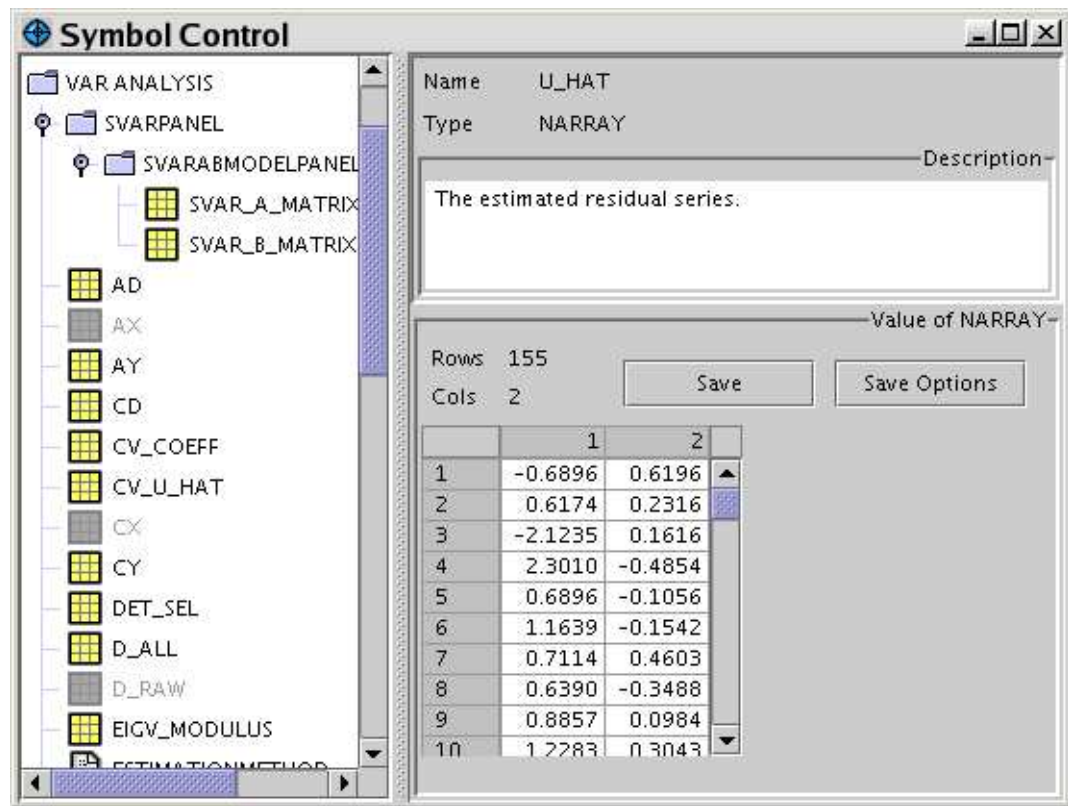


Figure 4.10: Screenshot of Symbol Control system for VAR model

model by offering information criteria for the choice of the optimal lag lengths, as well as more sophisticated subset search procedures, which automatically find zero restrictions in a model. Figure 4.8 presents the panel for applying model reduction strategies.

Once the model is estimated, see Figure 4.7, further analysis steps can be taken. But first, the model should be checked against various possible misspecifications with the help of the residual and the stability analysis panels. Also, the presence of ARCH effects can be analysed. The structural analysis can then be employed to convey an impulse response analysis, as well as a forecast error variance decomposition and causality tests. To identify and trace structural shocks, the SVAR analysis is available. Figure 4.9 shows the specification panel for the SVAR model estimation. Finally, forecasting of the levels and the undifferenced series is provided as an option to the user.

It should be noted that all analysis panels rely on results from the estimation, like the estimated residuals, the covariance matrix, etc.. Thus, the modelling steps are done separately with only one model being analysed at a time. The global symbol table of the Symbol Management system is used to share data that needs to be accessed by different panels. Figure 4.10 shows the symbols that are shared, as well as symbols that are local to the SVAR analysis panel. The variable that is currently selected is `U_HAT`, which holds the estimated residuals from the regression. Those residuals, for example, must be accessed by the panels for diagnostic tests to check against misspecifications in the estimated model. Data that only needs to be visible to one panel can be put in the local symbol tables to make it accessible via the Symbol Control system for data export. The variables `SVAR_A_MATRIX` and `SVAR_B_MATRIX` hold the matrices of restrictions on the structural parameters. They are only used by the SVAR analysis panel and therefore go in the local symbol table for the component with the name `SVARABMODEL PANEL`.

4.6.2 Implemented Features

Specification

All relevant parameters of Equation (4.2) can be selected with the help of the specification panel (Figure 4.6). Furthermore, zero restrictions on the coefficients can be set manually or by applying model reduction strategies with the subset specification panel (Figure 4.8).

Model Checking

To guard against misspecification, JMULTi provides a range of model checking tools that should be used to test whether assumptions of the underlying model are violated. Assumptions about the residuals can be checked with the Residual Analysis, parameter constancy can be investigated with the Stability Analysis. The ARCH Analysis may be used to model the volatility process of the estimated residuals.

- **Residual Analysis**
 - **Diagnostic tests** - A range of diagnostic tests can be applied to the estimated residuals of a VAR model. The Portmanteau test and the Breusch-Godfrey LM test (Edgerton and Shukur (1999)) check for remaining autocorrelation. Multivariate tests for nonnormality suggested by Doornik and Hansen (1994) and Lütkepohl (1991, Chapter 4) can be used. A multivariate ARCH-LM test is available as well (Doornik and Hendry (1997, Sec. 10.9.2.4)).
 - **Plot/Add** - The estimated residuals can be plotted and optionally be standardized or squared before. It is also possible to add them to the dataset again if they should be used as a variable for a new model.
 - **Correlation** - The AC and PAC functions can be plotted and printed for the residuals. It is also possible to compute the crosscorrelations of the residual series together with the exact asymptotic confidence intervals as described in Lütkepohl (1991), Sec. 4.4.2, for stable, unrestricted VARs and Sec. 5.2.9 for stable VARs with parameter constraints.
 - **Spectrum** - It is possible to show the spectrum of the available residuals, see also Section 4.5.2.
 - **Kernel Density** - For all estimated residual series, kernel density estimates can be computed and plotted or printed. It is also possible to compare the densities with that of a generated normal random variable.
- **Stability Analysis** - The assumption of parameter constancy may be checked with a number of different test procedures.
 - **Recursive parameter estimates** - Recursive parameter estimates are obtained by simply estimating the model using only data for $t = 1, \dots, \tau$ and letting τ vary from some small value to T , the end of the original sample. Here the same estimation method is used which is also used for the full sample estimation. The series of estimates together with

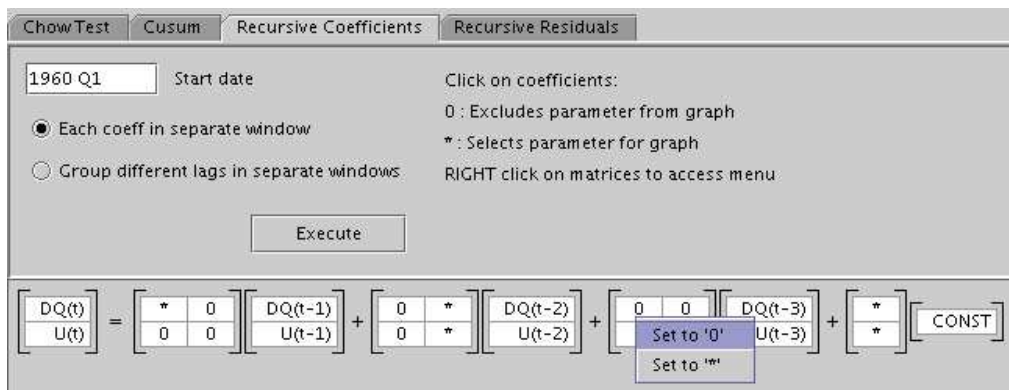


Figure 4.11: Screenshot of panel for plotting recursive coefficients estimates

two-standard error bands are then plotted and can convey useful information on the relative importance of new observations that are added to the sample. Because there may be many coefficients to be plotted, JMulTi provides an intuitive way to select single coefficients or matrices via mouse clicks or via a small popup menu, see Figure 4.11.

- **Recursive residuals** - The so-called recursive residuals are standardized 1-step forecast errors from a model estimated on the basis of data up to period $\tau - 1$. They are computed for the individual equations of a vector model separately. JMulTi allows to plot them and to adjust the coverage probability of the confidence intervals.
- **CUSUM tests** - JMulTi computes the CUSUM and CUSUM-of-squares tests (Brown et al. (1975)) and plots the resulting test statistics for each period together with the confidence intervals. If the test statistic falls outside the interval bounds, this is evidence against structural stability of the underlying model.
- **Chow tests** - Figure 4.12 shows the panel for specifying break-point (BP), sample-split (SS) and Chow forecast (CF) tests. It is either possible to compute the tests for a known breakpoint, or for all possible breakpoints within a given range. The p-values are computed by applying a bootstrap procedure (Candelon and Lütkepohl (2000)). In Figure 4.12 the tests were computed for a range of which every 6th

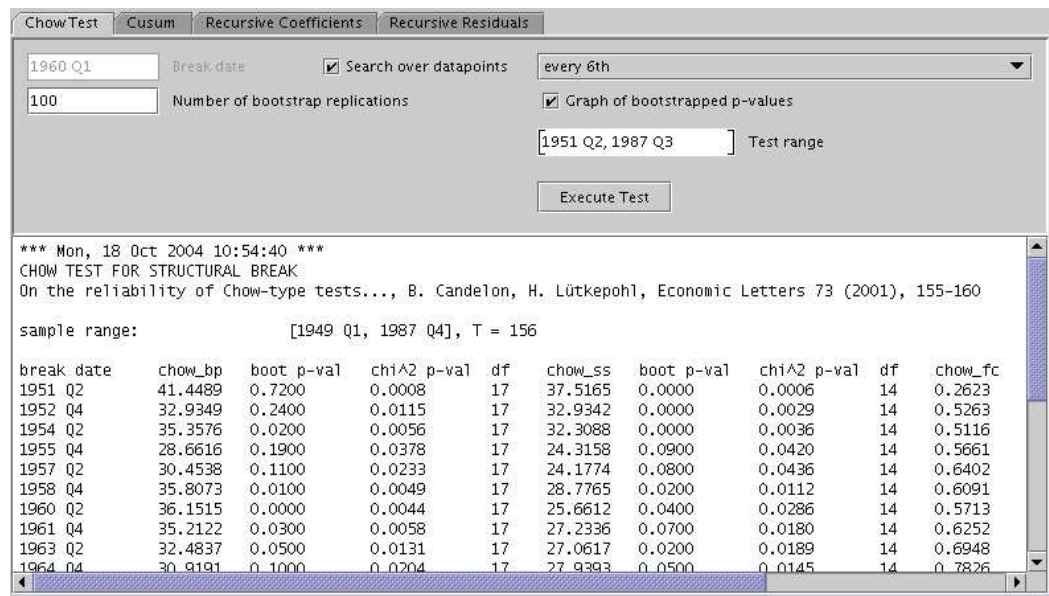


Figure 4.12: Screenshot of panel for computing bootstrapped Chow tests

date was considered as a break date. Because bootstrap procedures tend to be very time consuming, the selection of only a subset of all possible break dates may reduce the required computing time significantly.

- **ARCH Analysis** - The ARCH analysis module is available for the estimated residuals of VAR models, but it is also provided as a stand-alone module to model the volatility of selected series of a dataset directly. It is described in more detail in Section 4.8.

Forecasting

After a model was estimated the forecast panel is accessible from the menu, see Figure 4.13. Forecasts are based on conditional expectations, assuming independent white noise disturbances. Because in practise the true parameters of the VAR model are unknown, forecasts are based on the estimated parameters, see (Lütkepohl (1991, Sec. 3.5)).

It is possible to generate level forecasts given that the model was estimated in levels, as well as level forecasts given that the underlying series is in first differences. The values for the exogenous and deterministic variables have to be supplied for the forecast period, whereas for the endogenous variables the forecasts are used recursively. JMulTi uses a suitable heuristic to extrapolate the deterministic variables, which may contain a constant, a linear trend, and seasonal dummies. Shift dummies are set to the value they have at time T , whereas impulse dummies are set to 0 for all forecast periods. Exogenous variables cannot be extrapolated in a standardized way, therefore they have to be user specified. If the actual exogenous series from the dataset holds observations for the forecast period, these values are automatically used.

Specify Forecast Text (save/print)

Horizon: 10 CI coverage (%): 95 Variables to forecast: DQ, U

Deterministics:

	CONST	S1	S2	S3	TREND
1988 Q1	1	1	0	0	160
1988 Q2	1	0	1	0	161
1988 Q3	1	0	0	1	162
1988 Q4	1	0	0	0	163
1989 Q1	1	1	0	0	164
1989 Q2	1	0	1	0	165

Exogenous

Start date of plot (levels): 1960 Q1

Asymptotic CI Forecast Undifferenced Forecast Configure Undifferenced For...

Figure 4.13: Screenshot of forecast panel for VAR analysis

Structural Analysis

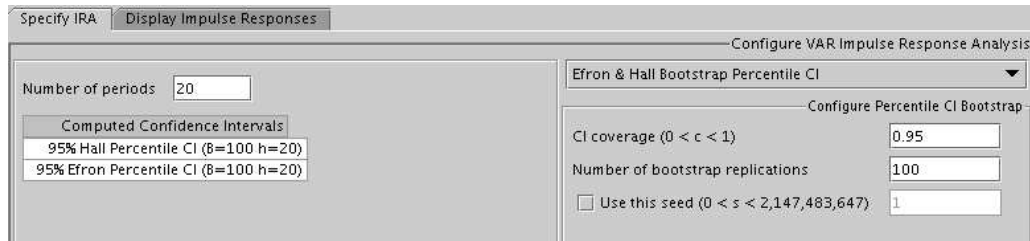


Figure 4.14: Screenshot of bootstrap specification panel for VAR Impulse Response Analysis

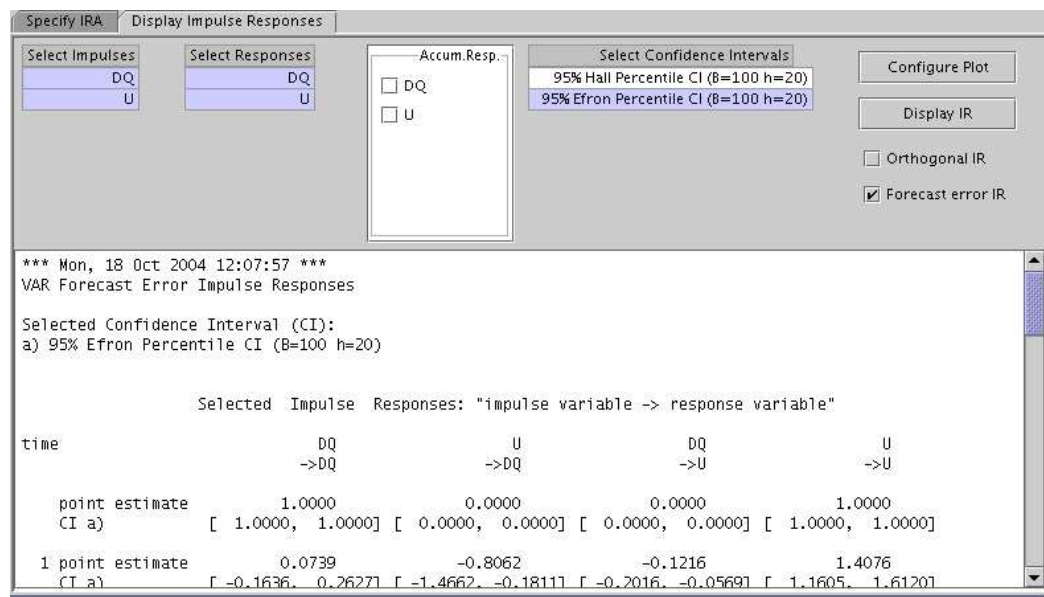


Figure 4.15: Screenshot of VAR Impulse Response Analysis panel

- **Causality Tests** - Two types of causality tests are implemented in JMulTi, tests for Granger-causality and tests for instantaneous causality. Those tests can be applied even before the VAR model is estimated, but after the variables, lag order and subset restrictions have been selected.

- **Impulse Response Analysis** - Impulse response analysis can be used to analyze the dynamic interactions between the endogenous variables of a VAR(p) process, treating the exogenous and deterministic variables as fixed. In JMulTi it is possible to create orthogonalized impulse responses based on an innovation of size one standard deviation in the transformed model as well as forecast error variance impulse responses based on a unit innovation in the original model. To construct confidence intervals around the computed impulse responses, JMulTi provides several bootstrap methods, which are the Standard percentile interval (Efron and Tibshirani (1993)), Hall's percentile interval (Hall (1992)), and Hall's studentized interval.

Because the bootstrap may be very time consuming and can also be adjusted with several options, it is presented in a separate panel, see Figure 4.14. The user can select the bootstrap method, the number of repetitions, the Confidence Interval (CI) coverage, as well as the seed. She might then choose an appropriate interval for the actual display from the generated bootstrapped CIs.

- **Forecast Error Variance Decomposition (FEVD)** - FEVDs are popular tools for analysing VAR models. They are interpreted as the percentage contribution of variable j to the h -step forecast error variance of variable k . In JMulTi one may select the variables which should be decomposed. The decomposition is presented as a plot, which displays the contributions of each variable to the FEV of the selected variable in a bar diagram. Text output is also provided.

SVAR

The SVAR (structural vector autoregressive) model may be used to identify the shocks that can then be traced in an impulse response analysis by imposing restrictions on the matrices **A** and **B** in the model form

$$Ay_t = A_1^*y_{t-1} + \cdots + A_p^*y_{t-p} + B_0^*x_t + \cdots + B_q^*x_{t-q} + C^*D_t + B\varepsilon_t \quad (4.3)$$

Here the structural errors ε_t are assumed to be white noise with $(0, I_K)$. The coefficient matrices are structural coefficients which may be different from the reduced form coefficients in (4.2). But the reduced form model has to be specified before the SVAR analysis can be applied. In the SVAR analysis only restrictions for A and B can be added. The reduced form residual u_t is recovered from the structural model as $u_t = A^{-1}B\varepsilon_t$ so that $\Sigma_u = A^{-1}BB'A^{-1}$.

- **AB-Model Estimation** - JMulTi offers three versions of the AB model, an A model where $B = I_K$, a B model where $A = I_K$, and a general AB model where restrictions can be placed on both matrices. Figure 4.9 shows the panel by which the restrictions can be set and which is used to start the ML estimation (Amisano and Giannini (1997)).
- **Blanchard-Quah Estimation** - In the Blanchard-Quah model $A = I_K$ and the matrix of long-run effects

$$(I_K - A_1 - \dots - A_p)^{-1}B$$

is assumed to be lower-triangular. If this model is chosen, restrictions are imposed automatically and the model can be estimated directly. If standard errors for the estimated long-run coefficients are desired, they can be computed by a bootstrap.

- **Impulse Response Analysis, FEVD** - Once an SVAR model was estimated the IRA and FEVD panels are activated and can be used to investigate the impact of the structural shocks.

4.6.3 Implementation Details

The VAR analysis module is a good showcase for the use of many subsystems of JStatCom in a demanding scenario with many panels, many variables and complex user interaction. This section will describe aspects of its implementation. Most of the solutions presented for the VAR module are also applied in the other analysis modules, which are VECM, STR, ARCH, and Nonparametric Analysis. Therefore the implementation sections for the remaining sections will be much

shorter, because basically the same problems had to be solved as in the VAR part. But this is also congruent with the idea of the underlying framework JStatCom, which tries to provide a design that can be applied to a wide range of modelling situations.

The VAR analysis module is significantly different from the Initial Analysis, because the availability of its panels depends on the actual modelling state. This is not the case for the Initial Analysis, because there only independent algorithms are applied and all panels are accessible at any time. As a contrast, in the VAR analysis the model estimation and the availability of the estimated parameters is a prerequisite for most subsequent modelling steps, like residual analysis, forecasting, etc.. Those panels are disabled until the required variables are not empty.

For this reason, the module must provide ways to adjust its behaviour according to the current state of the model. Furthermore, because there is always a single model being analyzed, there must be absolutely no ambiguity about which specification belongs to the currently displayed coefficients. This means that whenever a different model is specified by changing the variables, the sample, or the number of lags, all estimated parameters must be reset and the analysis must start from the beginning.

The described requirements have motivated some of the subsystems now being part of JStatCom, like the Symbol Event system. It soon became clear that the most consistent way to adjust the availability of certain modelling panels, mostly by enabling/disabling menu items, is achieved if the state of the required variables in the shared symbol table is checked. An alternative would have been to lay out specific ways to proceed in an analysis with a predefined sequence of steps, like in a wizard application. However, the current approach allows for more flexibility in the design of the user interface, as well as in the user interaction that is allowed for. The user can freely choose in which way he wants to proceed with the analysis, the module just makes sure that the requirements are met. Technically this is achieved via installing listeners to the symbols of interest.

The Symbol Event System

There is no specific class representing a VAR model in JMulTi. Instead, the module uses the global symbol table of the VAR frame to store all variables that are needed to represent a model. Therefore, changes in the model always correspond to changes in the data objects referenced by symbols. It is argued that every model can be represented by a set of symbols in a symbol table. Therefore this scheme is also used for all other modules, thus establishing a standard design which can be understood more easily than ad hoc solutions for each model class.

Consequently, if GUI components need to be in a consistent state with the model, they can use services provided by the Data Model, especially the Symbol Management (Section 3.10) and the Symbol Event system (Section 3.11). For example, the residual analysis depends on the availability of the estimated residual series, which are stored in the global symbol table of the VAR frame under the name `U_HAT`. The underlying data object is empty in the beginning and whenever a new model is specified. As soon as the VAR estimation is finished successfully, `U_HAT` is filled with the actual values and changes its state from *empty* to *not empty*. The best way to enable or disable the menu item *Residual Analysis* according to the empty state of `U_HAT` is to install a `SymbolListener` that listens to exactly those changes in that symbol. This listener is informed only when the empty state changes, and it can then change the state of the menu item accordingly. The following code example demonstrates this:

```
private JMenuItem resAn_menuItem;
...
SymbolListener listener = new SymbolListener(){
    public void valueChanged(SymbolEvent evt){
        resAn_menuItem.setEnabled(!evt.isSourceEmpty());
    }
}
global().getSymbol("U_HAT").addSymbolListener(listener,
    SymbolEventTypes.EMPTY_STATE);
```

Here it is assumed that the variable `resAn_menuItem` holds a reference to the menu item *Residual Analysis* of the VAR frame. A symbol listener is first created which sets the enabled property of `resAn_menuItem` to `true` if the symbol event

`evt` is not empty and vice versa. The created listener must then be registered with one or more symbols. In this case, the symbol `U_HAT` is retrieved from the global symbol table and the listener is installed. It should be noted that for performance reasons it is only informed when the empty state changes. Other changes are ignored.

A good place to install listeners that enable or disable menu items, and thereby access to modelling panels, is the module frame itself. The reason is that from within that class one typically has access to all menu items because they are declared there as `private` fields.

It can also be the case that the individual panels update their internal state according to the state of the shared symbols. They might install listeners themselves in the respective panel classes. For example, the SVAR panel updates the state of the restriction matrices whenever the dimension of the VAR process changes. However, such changes might as well be done when the component is actually being shown, because otherwise they are simply irrelevant. In this case a symbol listener would not be needed, but rather the `shown` method of `ModelPanel` could be overridden and a `CardPanelDisplay` could be used as the container for all modelling panels. Whenever a panel would be shown or hidden, the `shown` method of that panel would be called with a `boolean` parameter, thus appropriate actions could be taken when needed.

Handling many Variables

It has been mentioned that the VAR model is represented by a set of symbols in a global symbol table. It actually consists of 37 globally shared variables, most of them being required by several modelling panels. This number also includes symbols for the names of the variables belonging to different types, restriction matrices, the sample range with and without lag truncation, as well as the estimated parameters, etc.. It has been argued earlier, see Section 3.10.4, that if many variables are used it is strongly recommended to create a separate class with all type definitions. Those type definitions should be declared as `public static final` fields referencing `JSCTypeDef` instances. This way they can be accessed from all panels of a module frame.

For the users of the Gauss programming language, this type definition is somewhat similar to declaring global variables for a library in a `.dec` file to keep track of them. However, there is a significant difference here. In Gauss the global variables are directly declared, whereas in JStatCom this is just the type definition. Access to the symbols and the underlying data values can only be done if there is also access to the respective symbol table. Type definitions can be understood as keys which can be used with a shared data repository. All panels of a module frame have access to the globally shared symbol table for that module, but not panels from other modules, although all classes can access the type definitions because they are typically `public`. This restricts access to variables to classes that belong to a certain module. The code example shows parts of the class `VARConstants` that declares all type definitions for the shared variables:

```
public final class VARConstants {

    /**
     * Covariance matrix of estimated coefficients no divided by T.
     */
    public static final JSCTypeDef cv_par_Def = new JSCTypeDef(
        "cv_Coeff",
        JSCTypes.NARRAY,
        "Covariance matrix of estimated coefficients.");

    /**
     * The estimated residuals.
     */
    public static final JSCTypeDef u_hat = new JSCTypeDef(
        "u_hat",
        JSCTypes.NARRAY,
        "Estimated residuals.");
}
```

```
/**
 * Names of endogenous variables.
 */
public static final JSCTypeDef Ny_Def = new JSCTypeDef(
    "Ny",
    JSCTypes.SARRAY,
    "Names of endogenous variables.");

/**
 * Lag truncated endogenous data.
 */
public static final JSCTypeDef y_Def = new JSCTypeDef(
    "y",
    JSCTypes.NARRAY,
    "Lag truncated endogenous data.");

/**
 * Sample range with lag truncation.
 */
public static final JSCTypeDef T1_Def = new JSCTypeDef(
    "T1",
    JSCTypes.DRANGE,
    "Sample range with lag truncation.");
...

```

It should be noted that the programmer has to make sure that each name is unique, because symbols are stored under the name of the type definition. The naming conventions for variables in JStatCom must be followed (Section 3.8.1) and names are case insensitive. One might argue that one could as well have used the unique type definitions directly as keys without depending on the names being unique. From a pure programming perspective this would indeed have some advantages, but then referencing variables with names would not be possible anymore. But this is an intuitive way to start programming with the Symbol Management system without the need to have type definitions, which is completely appropriate for simple applications. The solution found is therefore a compromise between intuitive usage for *beginners* and more powerful concepts for *ad-*

vanced developers who need to handle heavy loads of variables. An example shows how globally shared variables can be accessed from within an instance of a `ModelPanel`:

```
// access via the type definitions
JSCDRange drange = global().get(VARConstants.T1_Def).getJSCDRange();
// access to the same symbol via the name (not recommended here)
JSCDRange drange1 = global().getSymbol("T1").getJSCDRange();
```

It is recommended to add a description to each type definition. If the definition is used to retrieve variables from a symbol table, the description is automatically set and displayed in the Symbol Control for the selected symbol. This greatly helps users to understand what is being stored in the symbols. It is also a good idea to add the description to the code comment for each variable, because it is then added to the JavaDoc documentation automatically when the javadoc tool is applied.¹

To summarize, having a separate class that holds `public static final` type definitions for the globally shared variables is always recommended. The name of that class can freely be chosen, but the suggested convention is to use the module name followed by the word `Constants`, for example `VARConstants`, `VECMConstants`, `STRConstants`, etc.. Because the type definitions represent the model being analysed, they should carefully be documented. It is of particular importance to know exactly what is stored in each variable to avoid confusion and to make maintenance and extensions more convenient. For example, it should be documented whether lags are specified for the levels or the 1st differences of the series, or whether a matrix with observations is already lag truncated or not. These small differences are often the cause of subtle programming errors that are hard to detect, and missing documentation is often the main cause for it. In the VEC analysis module, there are 84 variables being shared, and the importance of following these guidelines becomes even more obvious.

¹The homepage of the javadoc API documentation tool for the Java language is java.sun.com/j2se/javadoc.

Components

Having described how panels can react to changes in the model and how many variables should be dealt with, there are also some advanced GUI components which should help presenting output and gathering user input. For the Initial Analysis it has already been mentioned that the classes `TSSel`, `ResultField`, and `NumSelector` are frequently used. The same is true for all analysis modules, but there are also other systems being applied, in particular Data Table (Section 3.27) and Equation (Section 3.28).

Figure 4.13 shows an example of the use of the Data Table system for the forecast panel. It can be seen that the deterministic variables for the forecast period have been extrapolated and the values are shown in a `JSCNArrayTable`. It is possible to edit the presented values, because the user might want to adjust certain deterministic variables in special cases manually. Furthermore, the data table is embedded in a `JSCDataTableScrollPane` which manages the display of an appropriate row and column header. The row header shows the date and the column header the variable name for each series. `JStatCom` makes it relatively easy to setup and update this quite complex component.

```
public class ForecastPanel extends ModelPanel{
    private JSCNArrayTable detTable = null;
    private JSCDataTableScrollPane detScroll = null;
    ...

    private JSCNArrayTable getDetTable() {
        if (detTable == null) {
            detTable = new JSCNArrayTable();
            detTable.setPrecision(0);
            detTable.setSymbolScope(Scope.LOCAL);
            detTable.setSymbolName("DetForecast");
        }
        return detTable;
    }
}
```

```

private JSCDataTableScrollPane getDetScroll() {
    if (detScroll == null) {
        detScroll = new JSCDataTableScrollPane();
        detScroll.setRowHeaderShowing(true);
        detScroll.setColumnHeaderShowing(true);
        detScroll.setViewportView(getDetTable());
        detScroll.setSymbolScope(Scope.LOCAL);
        detScroll.setRowHeaderSymbolName("ForecastPeriod");
        detScroll.setColumnHeaderSymbolName("DetVarNames");
    }
    return detScroll;
}
// overwrites shown method that is called when this
// panel is shown or hidden in a CardPanelDisplay
public void shown(boolean isShown) {
    if (isShown) {
        JSCNArray det = new JSCNArray("DetForecast");
        //... compute deterministic for forecast period
        JSCSArray detNames = new JSCNArray("DetVarNames");
        //... determine names of deterministic
        JSCSArray time = new JSCNArray("ForecastPeriod");
        //... create array with date strings

        local().set(det);
        local().set(detNames);
        local().set(time);
    }
}
...
}

```

First, one should bear in mind that many of the options set in the methods `getDetScroll` and `getDetTable` are automatically generated if a visual composition editor is used. Thus manual programming is required only for the `shown` method. The deterministic data for the forecast is only updated when the compo-

ment is being displayed, therefore the `shown` method of `ModelPanel` was overwritten. It is called when a `CardDisplayPanel` is used as the container for all panels belonging to a certain analysis. This is just one way of doing it, one could as well install a symbol listener or invent other ways of triggering that computation.

By using `setSymbolScope` the setup of the data table and the embedding scrollpane was done in a way that the local symbol table is used. Therefore, if the data objects in that symbol table with the names "DetForecast", "ForecastPeriod", and "DetVarNames" change, the data table and the row and column headers of the scrollpane are automatically adjusted. For this reason, the `shown` method must create the respective data objects according to the model state and set them to the local symbol table via the method `local().set`.

In the same way, the restriction matrices for the SVAR estimation panels are set up and updated, see Figure 4.9. But there the default renderer of the data tables have been changed, the right mouse popup menu has been disabled, and a mouse click listener has been installed. The underlying code looks as shown in the following where the part for the update is omitted:

```
private JSCNArrayTable dataTableA = null;
private JSCDataTableScrollPane scrollPane = null;
...
private JSCNArrayTable getDataTableA() {
    if (dataTableA == null) {
        dataTableA = new JSCNArrayTable();
        dataTableA.setSymbolScope(Scope.LOCAL);
        dataTableA
            .setCellRenderer(JSCCellRendererTypes.DIAG_01M1NEGINF);
        dataTableA.setTablePopup(null);
        dataTableA
            .setMouseListener(JSCMouseListenerTypes.DIAG_01NEGINF);
        dataTableA.setEditable(false);
        dataTableA.setSymbolName("A_MATRIX");
    }
    return dataTableA;
}
```

```
private JSCDataTableScrollPane getDataTableScrollPaneA() {
    if (scrollPane == null) {
        scrollPane = new JSCDataTableScrollPane();
        scrollPane.setMinimumVisibleColumns(5);
        scrollPane.setMinimumVisibleRows(5);
        scrollPane.setViewportView(getDataTableA());
        scrollPane.setColumnHeaderShowing(true);
        scrollPane.setColumnHeaderSymbolName(VARConstants.Ny_Def.name);
    }
    return scrollPane;
}
```

In the method `getDataTableA` a data table is created with a special mouse click listener and an appropriate renderer. They are taken from the enumerations `JSCCellRendererTypes` and `JSCMouseListenerTypes`. Like in the previous example, the symbol scope is set to local and a name for the symbol is set. If the user changes the restrictions, the underlying symbol is automatically updated. The data table scrollpane is initialized in the method `getDataTableScrollPaneA`. It only shows a column header with the variable names, but no row header. This is disabled by default, but the column header is switched on via `setColumnHeaderShowing(true)`. The scrollpane also sets the name of the symbol that holds the variable names to display. Here it can be seen that the global symbol table is used and the shared variable definitions are referenced via `VARConstants.Ny_Def`. However, the method `setColumnHeaderSymbolName` does not take the type definition as an argument, but the actual name of that variable. Therefore the argument must be `VARConstants.Ny_Def.name`, which is a reference to the string "Ny" in that case.

Display of Equations

Figures 4.7, 4.8, and 4.11 show how VAR models are presented in a form that resembles the mathematical notation from Equation (4.2). This is useful to display complex output in a very intuitive way, as well as to gather user input that is directly related to the coefficients of a model. JStatCom provides the Equation

system (Section 3.28) for that purpose. It appears in the VAR module to present estimation output, to let the user select zero restrictions for the coefficients, as well as to select the coefficients for the recursive estimation. The Equation system is also used in the VEC and STR analysis modules.

Because equations are used for different tasks, the behaviour of the Equation system must be adjusted. To present the estimated coefficients, numbers have to be shown with the possibility to adjust the precision of the display via a popup menu. Editing is not allowed. For the selection of subset restrictions it is necessary to install a mouse listener that changes the values to the next valid value from a given set. At the same time a special cell renderer should be installed that paints numbers as symbols, for example a '*' to denote an unrestricted coefficient. Furthermore, a popup menu for changing the precision would not make sense in that context. Therefore one would like to disable it, or replace it with a menu allowing to change the values for whole matrices. JStatCom supports all these tasks with the Equation system, which is build on top of the Data Table system. Therefore the flexibility in setting arbitrary renderers, mouse listeners, and popup menus is also available for the equation components. A code example has been given in Section 3.28.4.

Input of Dates and Date Ranges

For some procedures user specified dates or date ranges have to be specified. Figure 4.12 shows the panel for specifying the Chow test for either a single date or all dates that fall in a certain range. JStatCom provides the two components `TSDateSelector` and `TSDateRangeSelector` for that purpose. They are described in Section 3.23. The panel for the Chow test is a good example on how to use these components, because typically the range that can legally be specified depends on the selected model. For this reason, the selection components must be dynamically adjusted to validate the user input according to the requirements of the underlying procedure. It usually does not make sense to set a static range that is used for input validation because it would need to change for every model. In the presented panel the range of the date selection component is updated whenever the component is shown. Therefore the `shown` method was overwritten. The panel is used inside a `CardPanelDisplay` which calls this method whenever a

child component is being shown or hidden. Parts of the actual implementation of that panel are presented in the following code:

```
public class ChowTestPanel extends ModelPanel {
    private TSDateRangeSelector testRange = null;
    private TSDateRangeSelector getTestRange() {
        if (testRange == null)
            testRange = new TSDateRangeSelector();
        return testRange;
    }
    public void shown(boolean isShown){
        if (!isShown)
            return;

        // the sample range
        TSDateRange range = global().get(VARConstants.T1_Def)
            .getJSCDRange().getTSDateRange();
        // number of observations
        int nr = range.numOfObs();
        // First possible: cols(y)*py + cols(x)*(px+1)
        //                + cols(d) + cols(y) + 1
        int py = global().get(VARConstants.py_Def).getJSCInt().intVal();
        int px = global().get(VARConstants.px_Def).getJSCInt().intVal();
        int K = global().get(VARConstants.y_Def).getJSCNArray().cols();
        // degrees of freedom are one less than 1st possible
        int degfree =
            py * K
            + (px+1)*global().get(VARConstants.x_Def).getJSCNArray().cols()
            + global().get(VARConstants.d_Def).getJSCNArray().cols()
            + K;
        // special case when py == 0
        degfree = (py == 0) ? ++degfree : degfree;
    }
}
```

```
// get the start date
TSDate start = range.lowerBound();
TSDate firstPossibleDate = start.addPeriods(degfree);
// creates new range
TSDateRange newRange = new TSDateRange(firstPossibleDate,
                                       nr - degfree - 1);
// sets range to selection component
getTestRange().setEnclosingRange(newRange);
getTestRange().setTSDateRange(newRange);
}
}
```

It can be seen that the date range selector is initialized in the method `getTestRange` without explicitly setting a validation range. The range is adjusted in the `shown` method which returns immediately if the component is hidden, because in that case `isShown` would be `false`. What follows is an algorithm that computes the first and last possible break dates and sets this as the enclosing range for the date range selection component. The formula for the required degrees of freedom is stated in the code comment. But to compute these dates for the selected model, the panel needs to retrieve all relevant information from the global symbol table. It uses the type definitions from the class `VARConstants` to reference the symbols from the global symbol table which is accessed via the `global` method. This way the lags, the dimension of the VAR model, the number of exogenous and deterministic variables, the number of observations, and the actual start date are retrieved and can be used to create a new `TSDateRange` object which is then used as the new enclosing range. It is also set to be the displayed range because often users want to apply the test over all possible break dates, therefore this is a reasonable default.

This more advanced example has shown how components might be used in a dynamically changing modelling environment. Although the underlying algorithms can get quite complex, it is argued that programming is made much easier with `JStatCom` by having a standard framework that can be applied to various different situations.

4.7 VEC Analysis

4.7.1 Overview

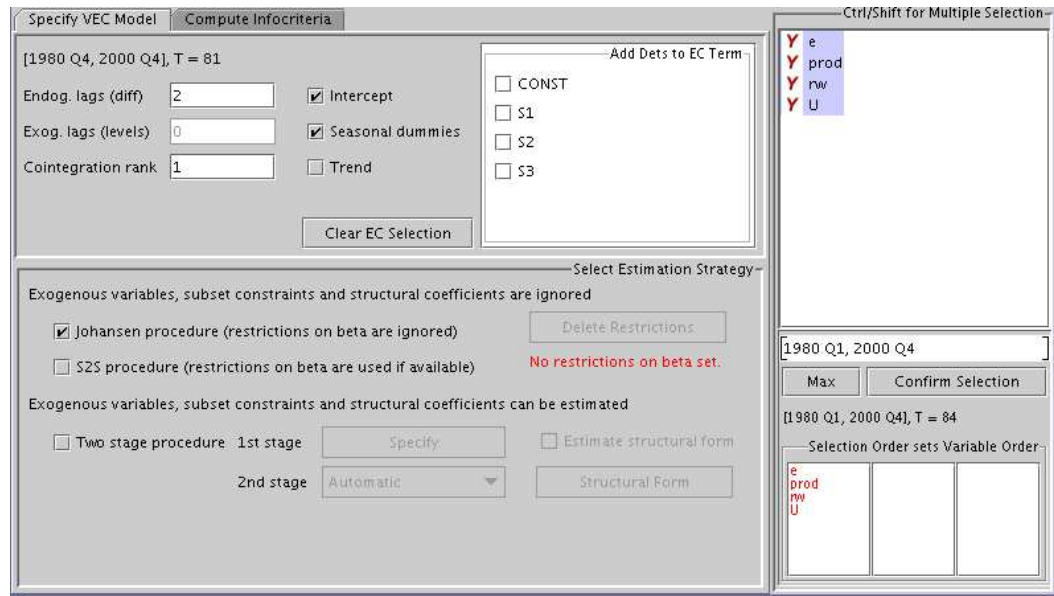


Figure 4.16: Screenshot of VEC model selection

The general setup of a VECM allowed for in JMulTi is of the form

$$\Gamma_0 \Delta y_t = \alpha [\beta' : \eta'] \begin{bmatrix} y_{t-1} \\ D_{t-1}^{co} \end{bmatrix} + \sum_{i=1}^p \Gamma_i \Delta y_{t-i} + \sum_{j=0}^q B_j x_{t-j} + C D_t + u_t, \quad (4.4)$$

where $y_t = (y_{1t}, \dots, y_{Kt})'$ is a vector of K observable endogenous variables, $x_t = (x_{1t}, \dots, x_{Mt})'$ is a vector of M observable exogenous variables, D_t^{co} contains all deterministic terms included in the cointegration relations and D_t contains all remaining deterministic variables. The Γ_i , B_j and C are parameter matrices with suitable dimensions. The residual vector u_t in Equation (4.4) is assumed to be a K -dimensional unobservable zero mean white noise process with positive definite covariance matrix $E(u_t u_t') = \Sigma_u$.

Figure 4.16 shows the specification panel for the VEC module. Deterministic variables may be constants, linear trends, seasonal dummy variables as well as

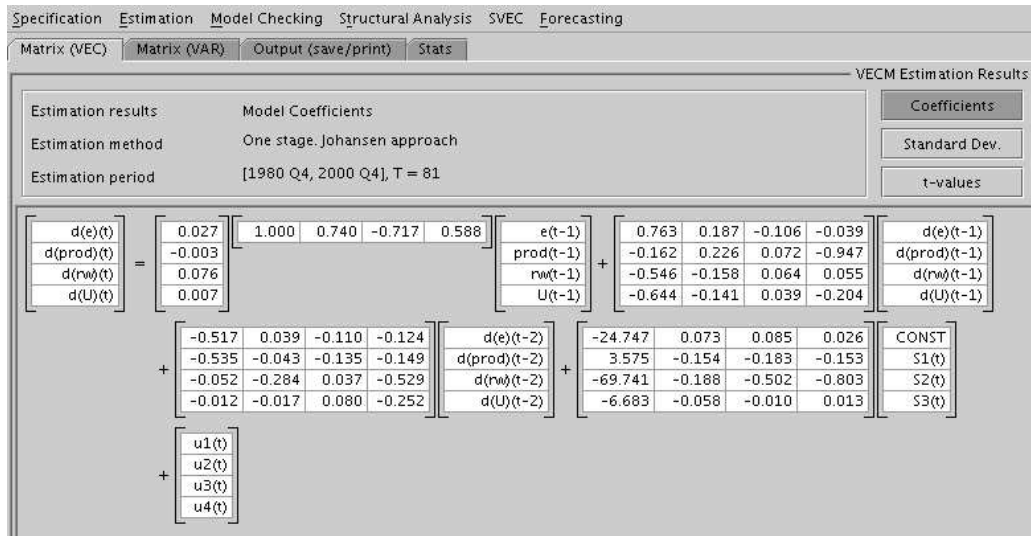


Figure 4.17: Screenshot of VEC estimation output in matrix form

user specified other dummy variables. It should be mentioned here that dummy variables can be created easily with the help of the TSSe1 component and the respective item in the popup menu of the time series list.

A single deterministic term cannot appear in both D_t and D_t^{co} so that the two vectors have to contain mutually exclusive terms. As can be seen in Figure 4.16 it is possible to select the deterministic variables that should appear only in D_t^{co} via a CheckBoxList. All selected deterministic variables appear in that list with their name.

The parameter matrices α and β have dimensions $(K \times r)$ and they have to have rank r . They specify the long-run part of the model with β containing the *cointegrating relations* and α representing the *loading coefficients*. The column dimension of η is also r . Like in the VAR module the estimated VEC model is presented to the user in matrix form which directly relates to the notation in Equation (4.4). This helps to avoid any ambiguities about the model that was actually estimated.

The cointegrating rank r has to be specified by the user. It must be in the range $1 \leq r \leq K - 1$. Cointegration tests for determining the cointegrating rank are available in the Initial Analysis part of JMulti, see Section 4.5.2. The number of

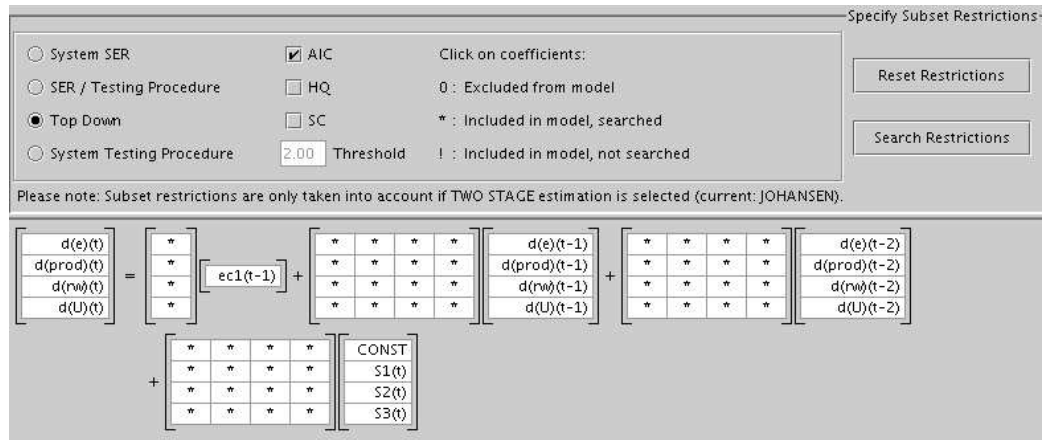


Figure 4.18: Screenshot of specifying restrictions on short-run dynamics for a VEC model

lagged differences of the endogenous variables, p , may be chosen with the help of model selection criteria.

Various restrictions can be imposed on the parameter matrices. In particular, it is necessary to impose restrictions to ensure an identified model form which can be estimated. Generally, (4.4) is a structural form which can only be estimated if identifying restrictions are imposed. If Γ_0 is specified to be an identity matrix, the model becomes a reduced form. It is possible to impose zero restrictions on the short-run parameters Γ_i , B_j and C , as well as on the loading coefficients α via the subset specification panel in Figure 4.18. Like in the VAR module it is possible to apply a model reduction strategy to find zero restrictions automatically.

Furthermore, it is possible to test and set restrictions on the cointegrating relations. The parameter matrices α and β have dimensions $(K \times r)$ and they have to have rank r . They specify the long-run part of the model with β containing the *cointegrating relations* and α representing the *loading coefficients*. The column dimension of η is also r and its row dimension corresponds to the dimension of D_t^{co} . The notation

$$\beta^* = \begin{bmatrix} \beta \\ \eta \end{bmatrix}$$

will be used in the following and the row dimension of β^* will be denoted by K^* .

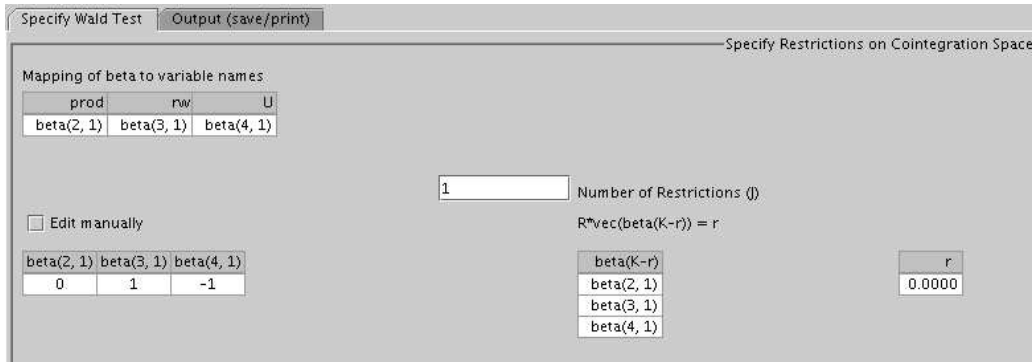


Figure 4.19: Screenshot of specifying restrictions on the cointegration relation of a VEC model

Hence, β^* is a $(K^* \times r)$ matrix. Restrictions on the cointegration relations may have the form

$$\text{vec}(\beta_{(K^*-r)}^*) = \mathcal{H}\eta + h \quad (4.5)$$

and can be estimated with the S2S procedure (Lütkepohl and Krätzig (2004, Chapter 3)). Here \mathcal{H} is a fixed matrix, h a fixed vector and η a vector of free parameters. These restrictions may be formulated alternatively as

$$R\text{vec}(\beta_{(K^*-r)}^*) = \mathbf{r}, \quad (4.6)$$

where R is a $(J \times (K^* - r)r)$ matrix and \mathbf{r} is a J -dimensional vector, as before. Figure 4.19 shows the panel that can be used to specify and test restrictions that are specified in the implicit form of Equation (4.6). For the actual estimation the implicit form must be converted to the explicit form defined by Equation (4.5). The conversion is done via the method `UData.imp2ExpRes` and takes R and \mathbf{r} as parameters, giving back an array with three `NArray` data objects holding \mathcal{H} , η and h . The `UData` class is part of `JStatCom` and has been mentioned in Section 3.20. This class also contains a few other helpful methods which should be checked with the help of the API documentation.

VEC modelling in `JMulti` is a step by step procedure, where each task is related to a special panel. Once a model has been estimated, the diagnostic tests as well as the stability analysis, structural analysis, SVEC estimation, and forecast-

ing use the results from the estimation. If changes in the model specification are made by the user, these results are deleted and the model has to be reestimated. In other words, only results related to one model at a time are kept in the system. Hence, there should be no confusion regarding the model setup while going through the analysis.

4.7.2 Implemented Features

This section lists the features that are part of the VEC module. The general structure is very similar to that of the VAR analysis, although the underlying algorithms are often different. However, many GUI specification panels look the same.

Specification

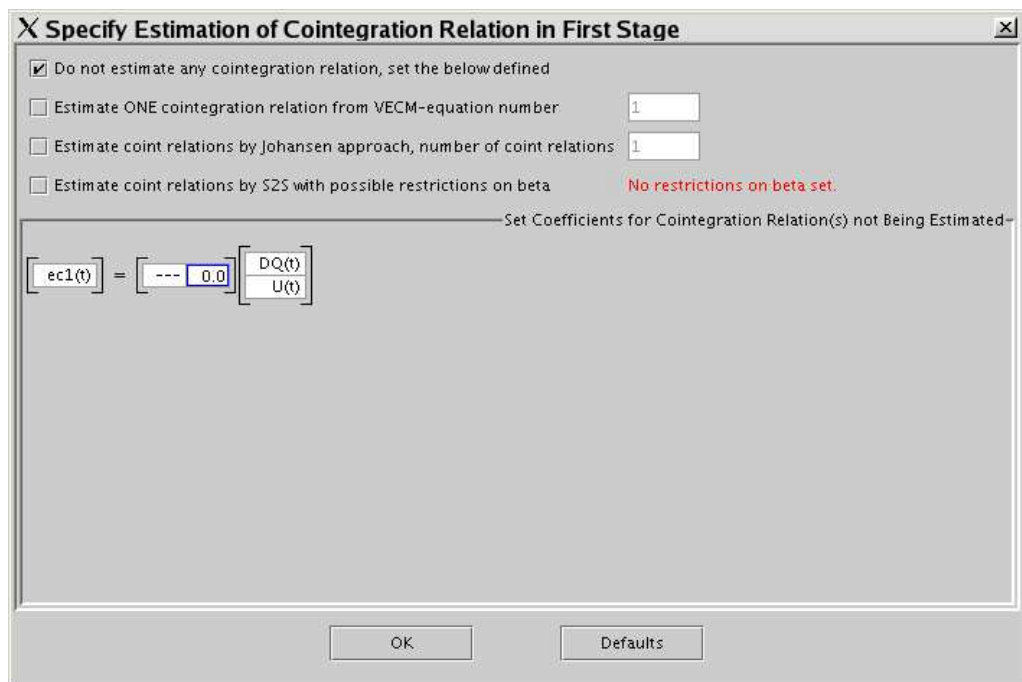


Figure 4.20: Screenshot of the dialog for specifying the estimation of the 1st stage of a two stage VEC estimation procedure

All parameters of the VEC model defined in Equation (4.4) can be set in the specification panel (Figure 4.16). Furthermore, restrictions on the long-run and short-run parameters may be set (Figures 4.18, 4.19). It is also possible to select the estimation procedure. The available options are Johansen, S2S, and two stage. For the two stage procedure one can further specify how the EC term is estimated via a dialog, see Figure 4.20. Options include to set the cointegration relation to a fixed vector or to estimate only parts of it. The specification panel also includes the option to compute the optimal endogenous lagged differences that are suggested by the information criteria.

Model Checking

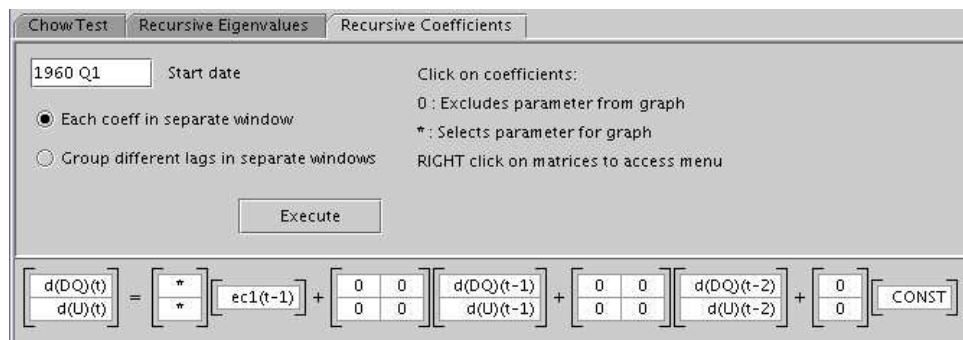


Figure 4.21: Screenshot of panel for plotting recursive coefficients estimates

Model checking in VEC models is quite similar to the implementation in VAR models. Assumptions about the residuals can be checked with the Residual Analysis, parameter constancy may be investigated with the Stability Analysis, and the multivariate ARCH Analysis can be applied to model the volatility process of the estimated residuals.

- **Residual Analysis -**

- **Diagnostic tests** - A range of diagnostic tests can be applied to the estimated residuals of a VEC model. The underlying algorithms are slightly different from the VAR case, but the specification panel looks exactly the same. The Portmanteau test and an LM test check for

remaining autocorrelation. Multivariate tests for nonnormality suggested by Doornik and Hansen (1994) and Lütkepohl (1991, Chapter 4) can be used. A multivariate ARCH-LM test is available as well (Doornik and Hendry (1997, Sec. 10.9.2.4)).

- **Further options** - Plot/Add, Correlation, Spectrum, Kernel Density behave exactly as in the VAR module, see Section 4.6.2.
- **Stability Analysis** - As in the VAR part the assumption of parameter constancy may be checked with a number of different test procedures.
 - **Recursive parameter estimates** - Figure 4.21 shows the panel for specifying the recursive parameter estimates of a VEC model. It looks very similar to Figure 4.11 for the VAR model, but here the EC term is part of the equation. It is possible to select recursive estimates for the loading parameters α as well.
 - **Recursive Eigenvalues** - For VEC models without parameter restrictions and without exogenous variables the eigenvalues from a reduced rank regression which are also used in the cointegration rank tests can be computed recursively by the Johansen estimation procedure (Johansen (1995)). Hansen and Johansen (1999) propose recursive statistics for the stability analysis of VEC models which are partly available in JMulTi.
 - **Chow tests** - The Chow test panel is similar to the one for the VAR analysis, except that for the test routine the VEC estimation procedure is used.
- **ARCH Analysis** - Multivariate ARCH analysis of the estimated residuals is possible for the VEC model. The ARCH analysis module is described in more detail in Section 4.8.

Forecasting

Forecasting with VEC models is based on the levels VAR form and is implemented in JMulTi as for VAR models.

Structural Analysis

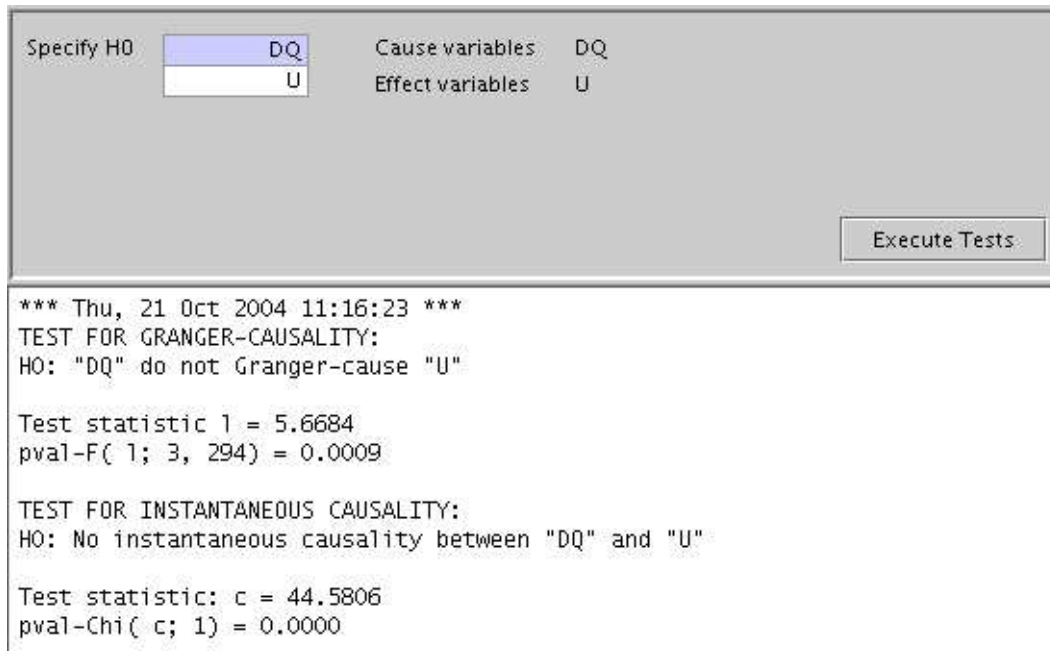


Figure 4.22: Screenshot of causality tests panel

- **Causality Tests** - Once a model without exogenous variables is specified, the causality analysis can be accessed. The endogenous variables are shown in the list. One can select from 1 up to $K - 1$ variables. The respective H_0 hypothesis then appears on the panel, see Figure 4.22. Tests for Granger-causality and tests for instantaneous causality are implemented.
- **Impulse Response Analysis, FEVD** - Impulse response analysis and FEVD of VEC models is based on the levels VAR form and is implemented in JMULTi as for VAR models, see Section 4.6.2.

SVEC

The SVEC (structural vector error correction) model can be used to identify the shocks to be traced in an impulse response analysis by imposing restrictions on

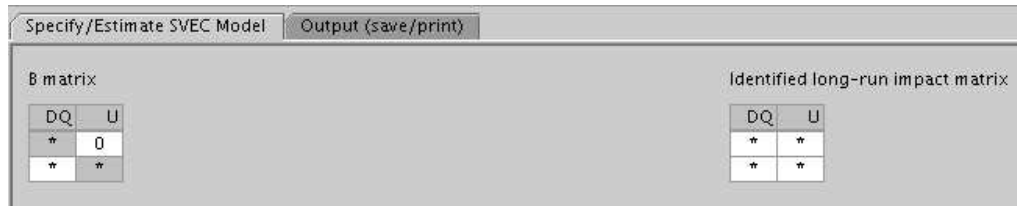


Figure 4.23: Screenshot of specification panel for SVEC estimation

the matrix of long-run effects of shocks and the matrix \mathbf{B} of contemporaneous effects of the shocks.

The matrix \mathbf{B} is defined such that $u_t = \mathbf{B}\varepsilon_t$ in (4.4) and, assuming that (4.4) is in reduced form, the matrix of long-run effects of the u_t residuals is

$$\Xi = \beta_{\perp} \left(\alpha'_{\perp} (I_K - \sum_{i=1}^{p-1} \Gamma_i) \beta_{\perp} \right)^{-1} \alpha'_{\perp}.$$

Hence, the long-run effects of ε shocks are given by

$$\Xi \mathbf{B}.$$

$rk(\Xi) = K - r$ and, hence, $\Xi \mathbf{B}$ has rank $K - r$. Thus, the matrix $\Xi \mathbf{B}$ can have at most r columns of zeros. Hence, there can be at most r shocks with transitory effects (zero long-run impact) and at least $k^* = K - r$ shocks have permanent effects. Due to the reduced rank of the matrix, each column of zeros stands for only k^* independent restrictions. $k^*(k^* - 1)/2$ additional restrictions are needed to exactly identify the permanent shocks and $r(r - 1)/2$ additional contemporaneous restrictions identify the transitory shocks. For examples see Breitung et al. (2004). JMulTi has a facility to impose restrictions on \mathbf{B} and $\Xi \mathbf{B}$, see Figure 4.23. The matrix on the left hand side sets the restrictions for \mathbf{B} , the one to the right is for restricting $\Xi \mathbf{B}$.

- **Impulse Response Analysis, FEVD** - Once a SVEC model was estimated the IRA and FEVD panels are activated and can be used to investigate the impact of the structural shocks.

4.7.3 Implementation Details

It has been mentioned in Section 4.6.3 of the VAR module description that basically the same problems that have been discussed there appear in all other analysis modules. As can be seen by comparing the screenshots presented for the VAR and VEC modules, there are many similarities between the implementations. For example, VEC modelling is also implemented as a step-by-step procedure where one model is analysed at a time. Thus the state of the panels depends on the state of the underlying model and the availability of estimation results. This is handled in the same manner as before by using the Symbol Management and Symbol Event systems.

However, listener management reaches a level of complexity that was dealt with by factoring out separate classes that implement the `SymbolListener` interface. For example, there is a class that listens to a number of symbols just to handle how menus are enabled and disabled. Another distinct listener class manages how the subset restriction matrices are assembled and reset according to the selected lags, the variables, and the cointegration rank. A third listener handles how observation matrices are put together because this depends on many settings, among which are the selection of deterministic variables that appear only in the EC term. It is also necessary to reset the estimation results whenever one of the model settings changes to avoid ambiguities. However, despite these quite challenging tasks, the Symbol Event system proved to be a sound basis for managing the required interaction between the GUI and the data model. It just required some more planning for the specific cases. For this, some experience with object-oriented programming is needed, one should especially be aware that classes do not acquire too many responsibilities. For example, simple listeners might be defined in some method of a model panel, but as soon as the listener itself gets large and has many different tasks to handle, one should consider writing a separate class for it. This is exactly what has been done for the VEC module. But it is important to note that the systems provided by JStatCom are still the basis for the design.

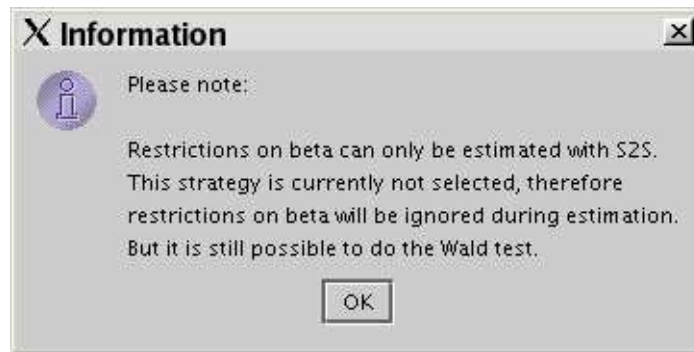


Figure 4.24: Screenshot of a user message about long-run restrictions not being taken into account with the currently selected estimation procedure

The VEC module is in many ways more complex than the VAR module, because there are more options to specify for the estimation and there are also different estimation procedures available. For certain specifications it is necessary to use a specific procedure, because otherwise it cannot be estimated in the desired way. For example, restrictions on long-run parameters are only taken into account if the S2S estimation option is set, either as a one-stage procedure or as the first stage of a two-stage estimation. Other estimation routines would ignore those restrictions. Furthermore, restrictions on the short-run parameters, as well as exogenous variables are only taken into account if a two-stage procedure is selected. The user still might want to compare the results of different estimation procedures, therefore adjusting those options automatically might lead to unwanted surprises. Instead, the user is informed with a text message (Figure 4.18) or a dialog box (Figure 4.24). However, user dialogs tend to interrupt the flow of interaction and should only be used in cases where it seems crucial to inform the user. In other cases one might consider text messages appearing in a status bar or in labels. Those messages can be ignored by the experienced user. This is done, for example, to indicate in the selection panel whether restrictions on the cointegration relations are set or not, see Figures 4.16 and 4.20.

Having more options for the estimation and a more complex model setup leads to a significant increase in the number of shared variables in the global symbol table, which is 84 instead of 37 for the VAR module. The need to define and document all variables carefully in the class `VECMConstants` is obvious in that

case. However, the system is able to handle this level of complexity, and it is easy to retrieve the needed estimation results for all panels that are used for subsequent modelling steps, for example

```
// gets the standard deviations of the estimated
// beta vector for the endogenous variables
global().get(VECMConstants.SD_beta_Def)

// gets the standard deviations of the estimated
// beta vector for the deterministic variables
// that are restricted to the EC term
global().get(VECMConstants.SD_beta_d_Def)
```

The Symbol Control provides another helpful tool to check what is actually stored. It can also be very useful for initial debugging.

By looking at the screenshots of the panels, one can also see several examples of the Equation system. It is used to present the estimation results in matrix form, to select subset restrictions on the short-run parameters, to set predefined coefficients for the cointegration relation (Figure 4.20), as well as to select the coefficients for the recursive estimation (Figure 4.21). In all cases, the equation terms have to be adjusted by setting special renderers, mouse listeners, and popup menus. The Data Table system is also used in various cases, for example to specify the restriction matrices for the SVEC estimation (Figure 4.23).

The intense use of these quite complex components demonstrates their usability for various tasks. It shows that software reuse with JStatCom can lead to a substantial gain in productivity for the programmer, and it is argued that the systems mentioned are flexible enough to be applied also in other problem domains than time series analysis. The need to present matrices with strings and numbers to the user appears in many applications that have to gather user input for complex algorithms.

It has been mentioned earlier that the package `de.jmulti.tools` contains components that are shared by different modules. Because the VAR and VEC modules have many user interface panels in common, it was worthwhile to design several components for shared use. However, because these panels need access to the global data of the estimated model, they must be told which model is used.

This is because the variable definitions for VAR and VEC models are different. Furthermore, some of those components are also used by other modules. The big advantage of sharing GUI classes is that they have to be laid out only once. Especially for complex GUIs this is a significant pay off. Another advantage is that reusing components tends to establish common standards for GUI design, because the user interfaces for similar tasks look exactly the same in all different modules. A deviation from that would certainly be considered *not* professional. All shared components are presented in Table 4.2.

Class Name	Responsibility
<code>de.jmulti.tools</code>	
<code>CausalityPanel</code>	Panel for causality tests.
<code>ChowTestPanel</code>	GUI for Chow test specification.
<code>CorrPanel</code>	Correlation analysis panel for estimated residuals.
<code>FEVDPanel</code>	Panel for computing the FEVD.
<code>ForecastPanel</code>	Forecast panel.
<code>ForecastUndiffConfig</code>	Dialog that is part of the forecast panel to specify forecasts of undifferenced series.
<code>IRABootCIPercPanel</code>	Panel for setting bootstrap options for Efron & Hall percentiles.
<code>IRABootCIStudPanel</code>	Panel for setting bootstrap options for studentized Hall percentiles.
<code>IRAComputePanel</code>	GUI for computing impulse responses and bootstrapped CIs.
<code>IRADisplayPanel</code>	Panel to manage plots and display of computed IR and CIs.

<code>KernDensPanel</code>	GUI for kernel density analysis, shared by all modules.
<code>MultDiagTests</code>	Panel for specifying multivariate diagnostic tests for VAR and VEC residual analysis.
<code>PlotAddResPanel</code>	Panel to add and plot estimated residuals.
<code>SpectrumPanel</code>	GUI for spectrum analysis.
<code>SubsetSpecPanel</code>	Panel to select model reduction strategy for VAR and VEC.

Table 4.2: Shared components

As a final remark, one should also notice that all procedure calls in the VEC module are encapsulated in subclasses of `PCall` and are therefore separated from the GUI components. All calls are put in the package `de.jmulti.proc`. Because there are many algorithms involved, it is important to maintain them in a standardized way. The `PCall` system (Section 3.19) provides the developer with the required services.

4.8 ARCH Analysis

4.8.1 Overview

A simple parametric model allowing for univariate modelling of time varying volatility is the ARCH(q) (Engle (1982)) process u_t with conditional variance σ_t^2 :

$$u_t = \xi_t \sigma_t, \quad \xi_t \text{ iid } N(0, 1), \quad (4.7)$$

$$\sigma_t^2 = \omega + \gamma_1 u_{t-1}^2 + \gamma_2 u_{t-2}^2 + \dots + \gamma_q u_{t-q}^2 \quad (4.8)$$

$$= z_t' \theta \quad (4.9)$$

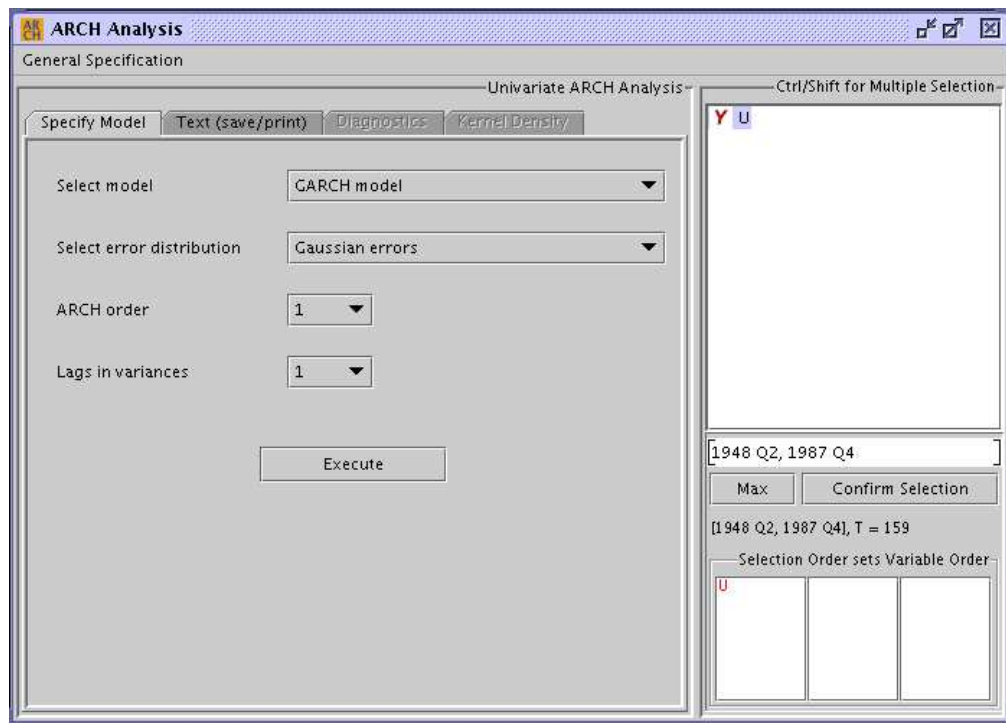


Figure 4.25: Screenshot of ARCH analysis module

The definition states that the second order moment of u_t is given conditional on an information set containing especially the history of the process. In the compact notation (4.9) $z_t = (1, u_{t-1}^2, \dots, u_{t-q}^2)'$ and $\theta = (\omega, \gamma_1, \dots, \gamma_q)'$. The $q + 1$ vector θ collects the parameters of interest. The generalization of the ARCH process is the so-called generalized ARCH (GARCH) process (Bollerslev (1986)). To allow for different impacts of lagged positive and negative innovations threshold GARCH models have been introduced by (Glosten et al. (1993)).

Multivariate GARCH models are a conceptually straightforward generalization of univariate models. Problems stem from the fact that a very large parameter space is involved, posing analytical and computational problems. The representation being in JMulTi is the BEKK form (Baba et al. (1990)) in its simplest form with $N = p = q = 1$. The estimation is implemented for a GARCH(1, 1) model, see (Herwartz (2004)) for a more detailed description.

4.8.2 Implemented Features

The ARCH module allows to specify univariate and multivariate models for the time varying volatility. They appear in different panels which can be selected via a menu from the ARCH module frame. The specification panel for univariate ARCH models is shown in Figure 4.25.

Univariate ARCH Models

```

*** Fri, 22 Oct 2004 11:20:58 ***
GARCH estimation with normal distributed errors for "xetra"
ARCH lags = 1
lags in variances = 1

variable      coefficient    t-statistic
intercept     6.05235e-02   2.35094
gamma(1)      9.43256e-02   5.98871
beta(1)       8.87922e-01  43.59559

covariance matrix
6.62771e-04   1.47690e-04  -4.01111e-04
1.47690e-04   2.48081e-04  -2.70468e-04
-4.01111e-04  -2.70468e-04  4.14824e-04

log likelihood = -1.98099e+03

```

Figure 4.26: Screenshot of output for univariate GARCH(1, 1) estimation

- Estimation** - In JMulTi the basic ARCH(q), GARCH(q, p) and TGARCH(q, p) models can be estimated up to orders $q = 5$ and $q = p = 2$, respectively. Maximum likelihood estimation can be performed under the assumptions of conditional normality, a conditional t -distribution or a conditional GED (Harvey (1990)). Consequently one has to specify the basic model, the lag-lengths, and the assumed conditional distribution.

The output (Figure 4.26) consists of the parameter values (where γ denotes the parameters of lagged errors and β denotes the parameters of lagged variances), the respective t -values, the variance-covariance matrix, and the value of the log likelihood function.

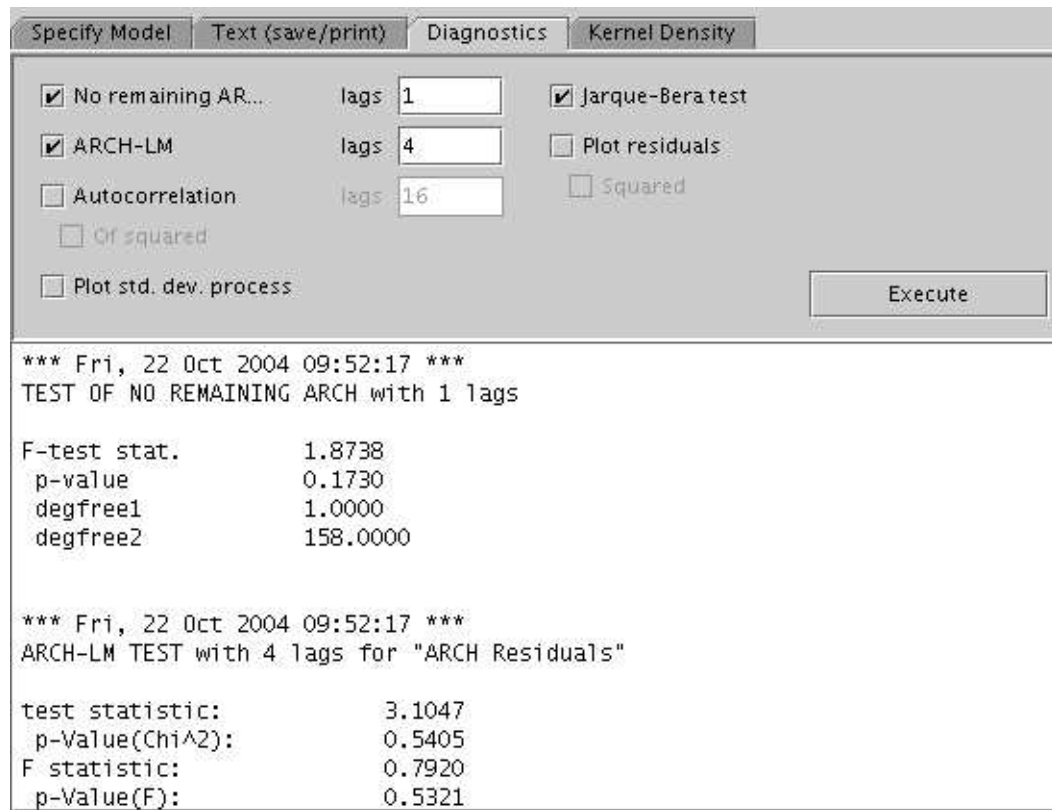


Figure 4.27: Screenshot of residual analysis for ARCH analysis

- Diagnostics** - The estimated residuals $\hat{\xi}_t$ can be analyzed to check whether the required assumptions are met, see Figure 4.27. Available tools are a test for no remaining ARCH (Lundbergh and Teräsvirta (2002)), the ARCH-LM test, plots of the AC and PAC functions of the squared residuals, Jarque-Bera tests for nonnormality, as well as plots of the estimated standard deviation process.
- Kernel Density Estimation** - Kernel density estimates can be done on the estimated residuals with a number of different kernels. It is also possible to compare the estimated densities with that of a generated normal random variable.

```

*** Fri, 22 Oct 2004 11:31:32 ***
Multivariate GARCH(1,1) estimation for "U_d1 DQ_d1"

Legend: estimation|(tvalues normal)

Gamma_0
1.68992e-01  7.71374e-01
( 0.41382 )  ( 0.23915 )
0.00000e+00  9.95948e-02
( 0.00000 )  ( 0.00488 )

Gamma_1
4.66318e-01  3.51523e-01
( 4.17116 )  ( 1.07866 )
-7.70019e-02  9.72467e-02
(-1.34046 )  ( 0.62259 )

Beta_1
6.91223e-01 -2.79318e-01
( 3.85395 )  (-0.49858 )
-1.08746e-01  7.68578e-01
(-0.39877 )  ( 1.36882 )

modulus of eigenvalues
5.71204e-01
8.63109e-01

```

Figure 4.28: Screenshot of output for multivariate GARCH(1, 1) estimation

Multivariate ARCH Models

Multivariate GARCH(1, 1) models can be specified for dimensions of 2, 3, and 4 variables. The model is estimated with a quasi maximum likelihood (QML) estimator under normality assumption. As an option it is possible to compute the exact QML t-ratios which require to evaluate the 1st and 2nd order derivatives of the Gaussian likelihood function analytically. The latter is left as an option to the user because the computation might be quite time consuming.

- **Estimation** - The output (Figure 4.28) consists of the parameter values (where gamma denotes the parameter matrices of lagged errors and beta denotes the parameter matrix of lagged variances), the respective t-values, and the value of the log likelihood function. Furthermore, the modulus of

the eigenvalues of the polynomial describing the unconditional mean of the covariance process are given to check for covariance stationarity.

- **Diagnostics** - To check whether the residuals meet the required assumptions, a number of multivariate diagnostic tests can be applied. Available are the Portmanteau test, the ARCH-LM test, plots of the AC and PAC functions of the residuals, Jarque-Bera tests for nonnormality, as well as plots of the estimated standard deviation process. It is also possible to plot estimated univariate GARCH(1, 1) processes together with the multivariate variance processes to analyse the differences.
- **Kernel Density Estimation** - Kernel density estimates can be done for each of the estimated residual series $\hat{\xi}_t$.

4.8.3 Implementation Details

The ARCH module is implemented in a similar way as the other analysis modules. Variables can be selected with the TSSel tool, estimation results are stored in the global symbol table, and all subsequent panels for model checking and graphical analysis are activated when the required data objects are changing their state to *not* empty. Interaction between the underlying model and the GUI is done via the Symbol Management and Symbol Event systems. The NumSelector component is used frequently for number input.

It should be noted that an alternative way to the number selection component provided by JStatCom is to use the standard component JComboBox to select valid values from a list. This can be seen in Figure 4.25, where the lags, as well as the error distributions and the model specification are selected with the help of this component. This is always appropriate if there is only a small number of available options. For the univariate ARCH module this is the case, because the number of lags is restricted.

It has been mentioned earlier that the ARCH module is not only available as a stand-alone frame, but also as part of the VAR and VEC modules to analyse the estimated residuals from those models. This is another example of internal component reuse. The panels holding the univariate and multivariate ARCH analysis

tools can also be used independently of the module frame. They might be put in all other analysis frames. In this case, there would be no TSSel component included, because the variables for the estimation are the estimated residuals which do not have to be selected. Of course, it is necessary to tell the panel the names of the required variables in the global symbol table that should be used as input for the ARCH estimation. This is done via setting properties of those panels. The rest of the analysis is handled internally within the ARCH panels via local symbol tables, because the residuals from the estimation of the volatility process are of no interest to any other panels and should therefore not be put in the global symbol table.

4.9 STR Analysis

4.9.1 Overview

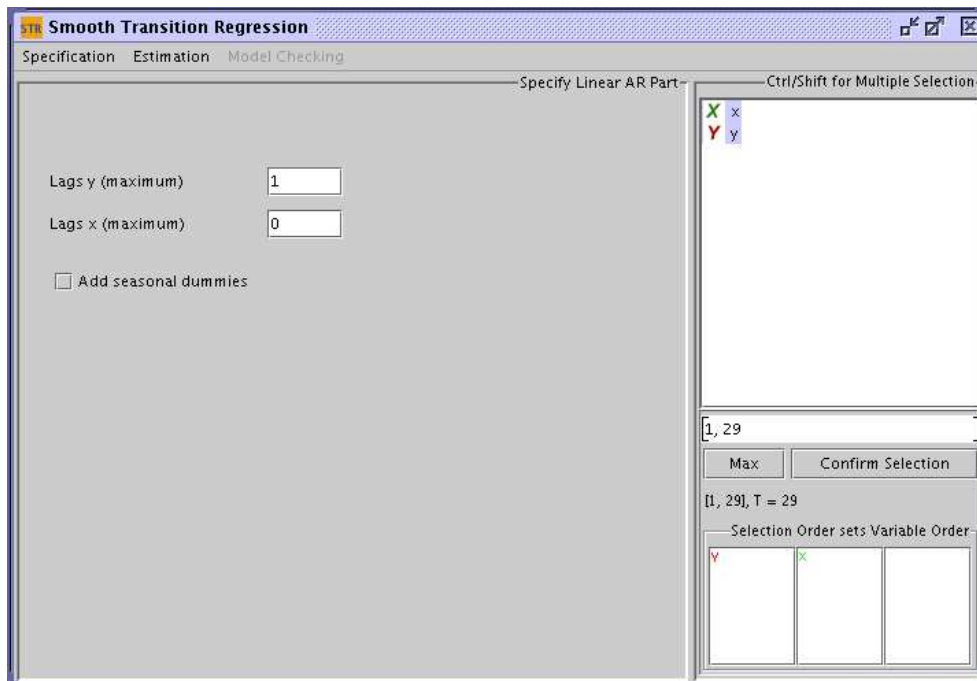


Figure 4.29: Screenshot of model selection for STR analysis

The standard STR model with a logistic transition function allowed for in JMulTi has the form

$$y_t = \phi' \mathbf{z}_t + \theta' \mathbf{z}_t G(\gamma, c, s_t) + u_t, \quad u_t \sim iid(0, \sigma^2)$$

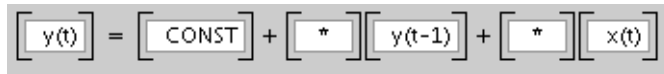
$$G(\gamma, c, s_t) = \left(1 + \exp\{-\gamma \prod_{k=1}^K (s_t - c_k)\} \right)^{-1}, \quad \gamma > 0$$

where $\mathbf{z}_t = (\mathbf{w}'_t, \mathbf{x}'_t)'$ is an $((m + 1) \times 1)$ vector of explanatory variables with $\mathbf{w}'_t = (1, y_{t-1}, \dots, y_{t-p})'$ and $\mathbf{x}'_t = (x_{1t}, \dots, x_{kt})'$. ϕ and θ are the parameter vectors of the linear and the nonlinear part respectively. The transition function $G(\gamma, c, s_t)$ depends on the transition variable s_t , the slope parameter γ and the vector of location parameters c . In JMulTi only the most common choices for K are available, $K = 1$ (LSTR1) and $K = 2$ (LSTR2). The transition variable s_t can be part of \mathbf{z}_t or it can be another variable, like for example $s_t = t$ (trend).

4.9.2 The Modelling Cycle

The modelling cycle suggested by the implementation in JMulTi consists of the three stages: specification, estimation and evaluation.

1. Specification starts with setting up a linear model that forms a starting point for the analysis. It can be modelled by using the VAR framework. The second part of the specification involves testing for nonlinearity, choosing s_t and deciding whether LSTR1 or LSTR2 should be used.
2. Estimation involves finding appropriate starting values for the nonlinear estimation and estimating the model.
3. Evaluation of the model usually includes graphical checks as well as various tests for misspecification, such as error autocorrelation, parameter nonconstancy, remaining nonlinearity, ARCH and nonnormality.



$$y(t) = \text{CONST} + * y(t-1) + * x(t)$$

Figure 4.30: Screenshot of selecting subset restrictions for AR parts of STR model

4.9.3 Implemented Features

Specifying the AR Part

The first step of the specification of an STR model is to select the linear AR model to start from. The selection mechanism allows to choose one endogenous variable y_t and an arbitrary number of exogenous (\mathbf{x}_t) and deterministic variables, see Figure 4.29. The maximum lag order for y_t and \mathbf{x}_t determines the number of lags to include. Lags of y_t and \mathbf{x}_t can be excluded again via setting subset restrictions as is shown in Figure 4.30. A constant is always included, seasonal dummies can be added via a button. Because the linear part must be stationary, there is no option to include a trend.

Transition Variable The transition variable s_t must be part of the selected variables or lags of these variables if it is not a trend. If s_t is not part of \mathbf{z}_t , then it can be excluded from \mathbf{z}_t via setting subset restrictions. One can also set $s_t = t$ in later stages, in that case it does not have to be included.

Smooth Trend It is possible to specify a smooth trend model

$$y_t = \phi_0 + \theta_0 G(\gamma, c, t) + u_t$$

by just including y_t with zero lags and specifying TREND as s_t in the estimation. The subset panel allows to impose exclusion restrictions on the selected variables by just clicking on the corresponding parameter and setting it to 0. To find those restrictions one could apply a subset search routine implemented in the VAR model part of JMulTi.

The excluded variables can still be used as transition variables but they are excluded from both, the linear and nonlinear parts, by setting $\phi_i = \theta_i = 0$. Other restrictions on the parameter space can be set in later stages of the analysis.

Testing Linearity

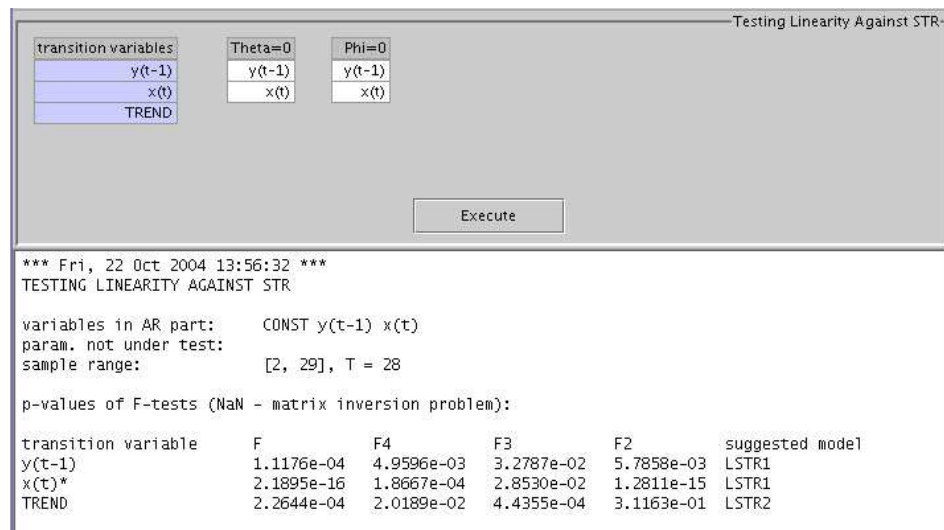


Figure 4.31: Screenshot of test for nonlinearity

The test can be used to check, whether there is a nonlinearity of the STR type in the model (Figure 4.31). It also helps to determine the transition variable and whether LSTR1 or LSTR2 should be used. The following auxiliary regression is applied if s_t is an element of \mathbf{z}_t :

$$y_t = \beta'_0 \mathbf{z}_t + \sum_{j=1}^3 \beta'_j \tilde{\mathbf{z}}_t s_t^j + u_t^* \quad (4.10)$$

whith $\mathbf{z}_t = (1, \tilde{\mathbf{z}}_t)'$. In case s_t is not part of \mathbf{z}_t :

$$y_t = \beta'_0 \mathbf{z}_t + \sum_{j=1}^3 \beta'_j \mathbf{z}_t s_t^j + u_t^*$$

The null hypothesis of linearity is $H_0 : \beta_1 = \beta_2 = \beta_3 = 0$. In JMulTi this linear restriction is checked by applying an F test. It is denoted in the output with the F symbol.

All potential transition variables can be selected in the respective table. The test is then executed for each of the candidates and the variable with the strongest test rejection (the smallest p -value) is tagged with the symbol *. This can be used as a decision rule for choosing an appropriate s_t , especially if the differences are big.

It is also possible to carry out the linearity test under further restrictions about θ . A variable can be excluded from the nonlinear part if $\theta_i = 0$. In the linearity test this can be taken into account by setting elements of $\beta_j = 0$. In JMulTi this can be done by selecting elements from the respective table. If all variables are excluded from the nonlinear part, then the test cannot be computed for s_t that are part of \mathbf{z}_t . However, it still works for transition variables not contained in \mathbf{z}_t , for example t .

Once linearity has been rejected, one has to choose whether an LSTR1 or an LSTR2 model should be specified. The choice can be based on the test sequence:

1. test $H_{04} : \beta_3 = 0$
2. test $H_{03} : \beta_2 = 0 | \beta_3 = 0$
3. test $H_{02} : \beta_1 = 0 | \beta_2 = \beta_3 = 0$

The test is based on the same auxiliary regression (4.10) as the linearity test. In JMulTi the corresponding F -statistics of the null hypothesis H_{04}, H_{03}, H_{02} are denoted by $F4, F3, F2$. The chosen model is explicitly stated. However, if the test sequence does not provide a clear-cut choice between the alternatives, it is sensible to fit both, an LSTR1 and LSTR2 model and decide on the evaluation stage. This can be done by looking at the information criteria or at the residual sums of squares.

Finding Initial Values

The parameters of the STR model are estimated by a nonlinear optimization routine. For the algorithm to work it is important to use good starting values. The

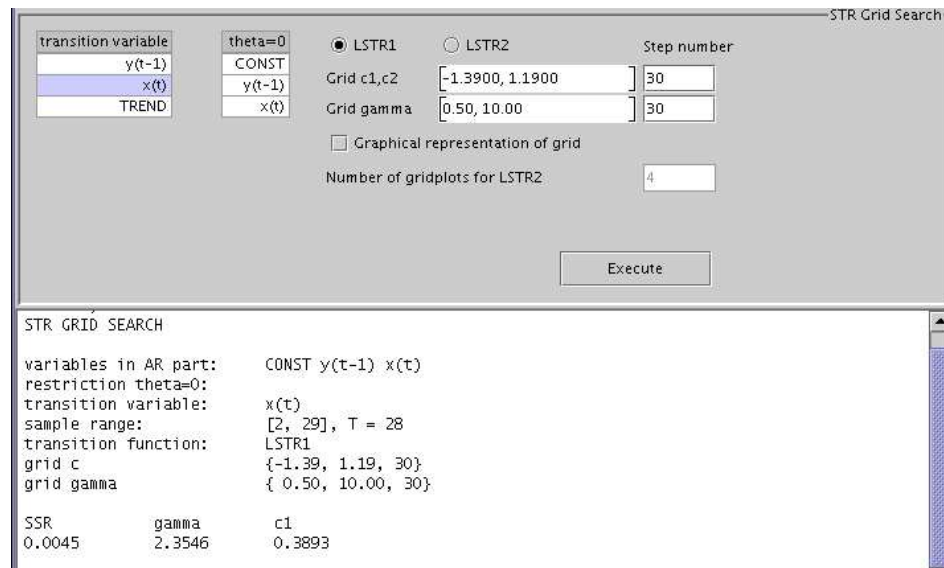


Figure 4.32: Screenshot of grid search to find starting values

gridsearch (see Figure 4.32) creates a linear grid in c and a log-linear grid in γ . For each value of γ and c the residual sum of squares is computed. The values that correspond to the minimum of that sum are taken as starting values. It should also be noted that in order to make γ scale-free, it is divided by $\hat{\sigma}_s^K$, the K th power of the sample standard deviation of the transition variable.

Estimation of Parameters

Once good starting values have been found, the unknown parameters c , γ , θ , ϕ can be estimated by using a form of the Newton-Raphson algorithm to maximize the conditional likelihood function.

The following types of restrictions are available for estimation:

1. $\theta_i = -\phi_i$ the corresponding parameter will disappear if $G(\gamma, c, s_i) = 1$
2. $\phi_i = 0$ the corresponding parameter will disappear if $G(\gamma, c, s_i) = 0$
3. $\theta_i = 0$ the variable will only enter the linear part

A variable can only be selected for one of those restrictions. Figure 4.34 shows the dialog for setting the restrictions for the estimation.

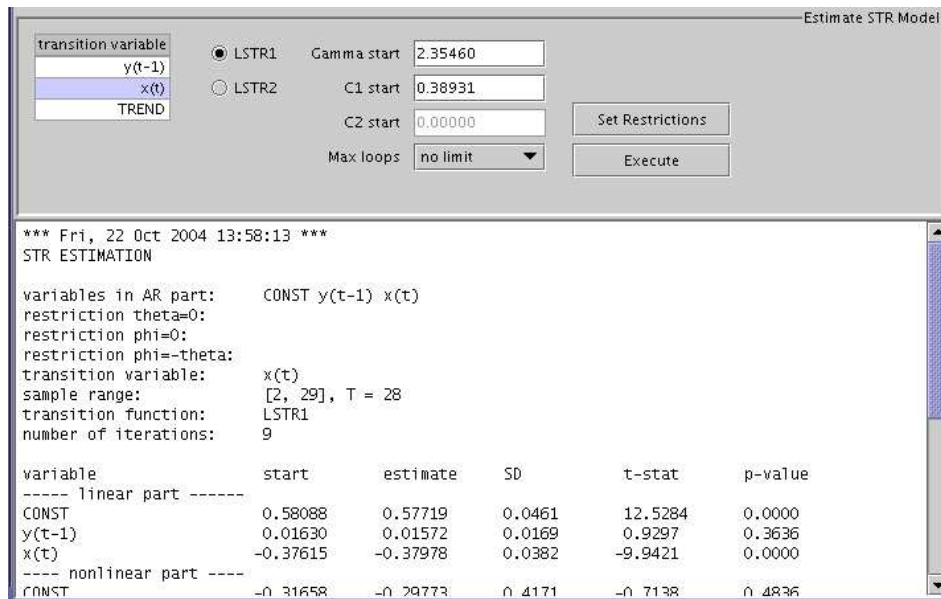


Figure 4.33: Screenshot of panel for STR estimation

The starting values, parameter estimates, standard deviations, t-values and p-values are displayed for the estimated model separated for the linear and the non-linear part. The covariance is computed by

$$\hat{\Sigma}_{\hat{\beta}} = 2\hat{\sigma}^2 [t] \left[\frac{\delta^2 S}{\delta\beta\delta\beta'} \right]^{-1} \Big|_{\hat{\beta}}$$

The following statistics are given as well:

1. AIC, SC, HQ
2. R^2 and adjusted R^2
3. variance and standard deviation of s_t
4. variance and standard deviation of estimated residuals

Figure 4.33 shows the panel holding the GUI to specify the estimation, as well as the output from the computations.

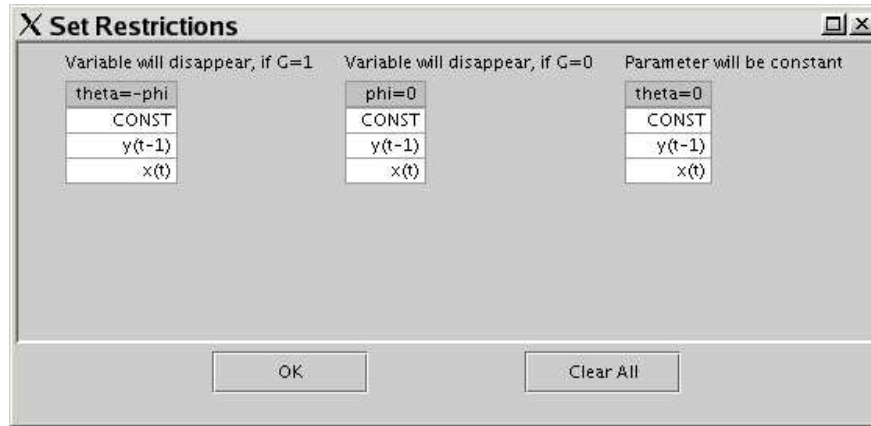


Figure 4.34: Screenshot of dialog to set restrictions for STR estimation

Graphical Analysis

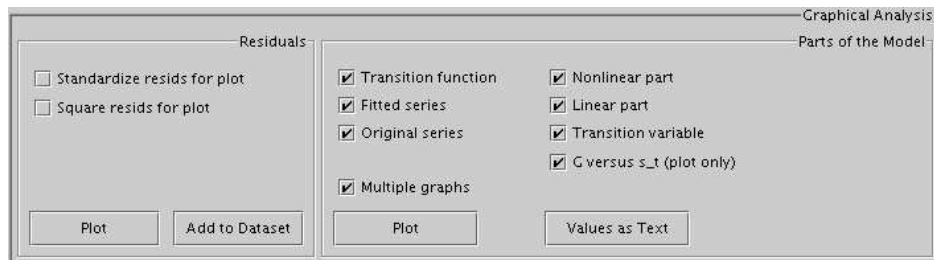


Figure 4.35: Screenshot of graphical analysis panel

Various parts of the estimated STR model can easily be plotted (Figure 4.35). This can help to detect problems in the residuals quickly and to illustrate the estimation results.

The estimated residuals

$$\tilde{u}_t = y_t - \tilde{\phi}' \mathbf{z}_t + \tilde{\theta}' \mathbf{z}_t G(\tilde{\gamma}, \tilde{c}, s_t)$$

can be plotted in various ways. It is also possible to add them to the dataset. They will appear as a new variable with the name `str_resids` in the time series list.

It is possible to plot against t :

1. the transition function $G(\tilde{\gamma}, \tilde{c}, s_t)$
2. the fitted series $\tilde{\phi}'\mathbf{z}_t + \tilde{\theta}'\mathbf{z}_t G(\tilde{\gamma}, \tilde{c}, s_t)$
3. the original series y_t
4. the nonlinear part $\tilde{\theta}'\mathbf{z}_t G(\tilde{\gamma}, \tilde{c}, s_t)$
5. the linear part $\tilde{\phi}'\mathbf{z}_t$
6. the transition variable s_t

It is also possible to plot the transition function as function of transition variable:

$$G(s_t) = \left(1 + \exp\left\{-\tilde{\gamma} \prod_{k=1}^K (s_t - \tilde{c}_k)\right\} \right)^{-1}$$

with $K = 1$ for LSTR1 or $K = 2$ for LSTR2.

Checking Misspecification

The quality of the estimated nonlinear model should be checked against misspecification like in the linear case, see Figure 4.36. The tests for STR models are generalizations of the corresponding tests for misspecification in linear models.

Test for No Error Autocorrelation The test is a special case of a general test described in Godfrey (1988) and has been discussed in its application to STR models in Teräsvirta (1998). The procedure is to regress the estimated residuals \tilde{u}_t on lagged residuals $\tilde{u}_{t-1} \dots \tilde{u}_{t-q}$ and the partial derivatives of the log-likelihood function with respect to the parameters of the model. The test statistic is then

$$F_{LM} = \{(SSR_0 - SSR_1)/q\} / \{SSR_1/(T - n - q)\}$$

where n is the number of parameters in the model, SSR_0 the sum of squared residuals of the STR model and SSR_1 the sum of squared residuals from the auxiliary regression.

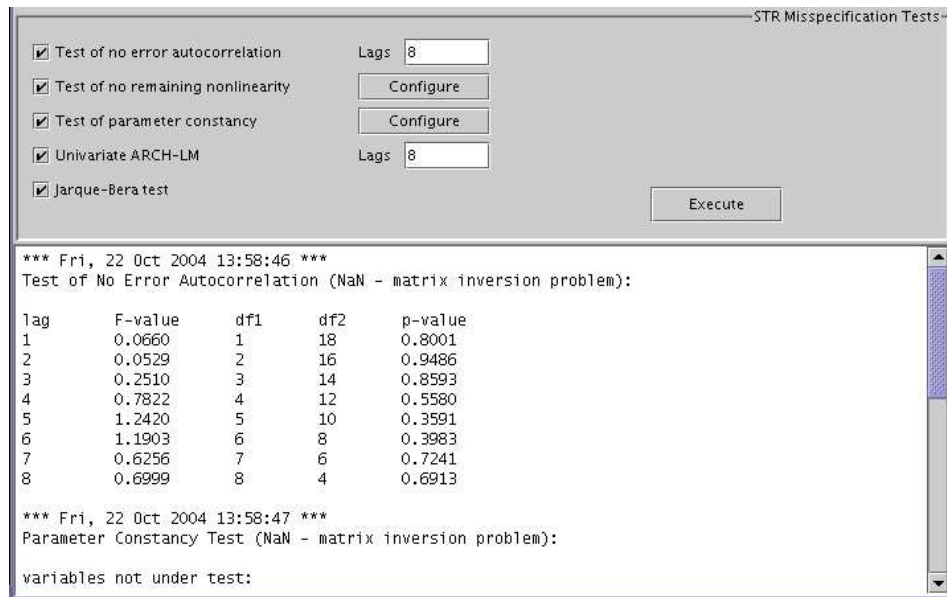


Figure 4.36: Screenshot of residual analysis panel

Test for No Remaining Nonlinearity After the STR has been fitted, it should be checked whether there is remaining nonlinearity in the model. The test assumes that the type of the remaining nonlinearity is again of the STR type. The alternative can be defined as:

$$y_t = \phi' \mathbf{z}_t + \theta' \mathbf{z}_t G(\gamma_1, c_1, s_{1t}) + \psi' \mathbf{z}_t H(\gamma_2, c_2, s_{2t}) + u_t$$

where H is another transition function and $u_t \sim iid(0, \sigma^2)$. To test this alternative the auxiliary model:

$$y_t = \beta'_0 \mathbf{z}_t + \theta' \mathbf{z}_t G(\gamma_1, c_1, s_{1t}) + \sum_{j=1}^3 \beta'_j \tilde{\mathbf{z}}_t s_{2t}^j + u_t^*$$

is used. The test is done by regressing \tilde{u}_t on $(\tilde{\mathbf{z}}_t s_{2t}, \tilde{\mathbf{z}}_t s_{2t}^2, \tilde{\mathbf{z}}_t s_{2t}^3)'$ and the partial derivatives of the log-likelihood function with respect to the parameters of the model. The null hypothesis of no remaining nonlinearity is that $\beta_1 = \beta_2 = \beta_3 = 0$. The choice of s_{2t} can be a subset of available variables in \mathbf{z}_t or it can be s_{1t} . It is also possible to exclude certain variables from the second nonlinear part by

restricting the corresponding parameter to zero. The resulting F statistics are given in the same way as for the test on linearity.

Test for Parameter Constancy This is a test against the null hypothesis of constant parameters against smooth continuous change in parameters. The alternative can be written as follows:

$$y_t = \phi(t)' \mathbf{z}_t + \theta(t)' \mathbf{z}_t G(\gamma, c, s_t) + u_t, \quad u_t \sim iid(0, \sigma^2)$$

where

$$\phi(t) = \phi + \lambda_\phi H_\phi(\gamma_\phi, c_\phi, t^*)$$

and

$$\theta(t) = \theta + \lambda_\theta H_\theta(\gamma_\theta, c_\theta, t^*)$$

with $t^* = t/T$ and $u_t \sim iid(0, \sigma^2)$. The null hypothesis of no change in parameters is $\gamma_\theta = \gamma_\phi = 0$. The parameters γ and c are assumed to be constant. The following nonlinear auxiliary regression is used:

$$y_t = \beta_0' \mathbf{z}_t + \sum_{j=1}^3 \beta_j' \mathbf{z}_t (t^*)^j + \sum_{j=1}^3 \beta_{j+3}' \mathbf{z}_t (t^*)^j G(\gamma, c, s_t) + u_t^*$$

In JMulTi the F-test results are given for three alternative transition functions

$$H(\gamma, c, t^*) = \left(1 + \exp\{-\gamma \prod_{k=1}^K (t^* - c_k)\} \right)^{-1} - \frac{1}{2}, \quad \gamma > 0$$

with $K = 1, 2, 3$ respectively and assuming that $\gamma_\theta = \gamma_\phi$.

Furthermore, the ARCH-LM and Jarque-Bera tests are available.

4.9.4 Implementation Details

The STR module again has a similar design to the other analysis modules. The Equation system is used to select subset restrictions (Figure 4.30). A challenging task for the STR module was to provide intuitive selection mechanisms for the transition variable, as well as for the various possible restrictions. For this, the

Data Table system was applied extensively with `JSCSArrayTable` instances, see Figures 4.31, 4.34, 4.32, 4.33. It is also applied in two more dialogs for the diagnostic tests that are not shown. Those tables show the variables that are eligible for selection by their names. The underlying logic of the module manages how the `SARRAY` data objects with the names are actually assembled according to the selected model. Although it required some effort to develop the needed algorithms for handling all possible selections and to translate them to the correct indices which can be handled by the underlying Gauss algorithms, it should be noted that also in this case the Symbol Management and Symbol Event systems provide the basis for triggering these algorithms and for exchanging data objects between various panels.

Because there are as many as 36 globally shared variables, it was again necessary to use a separate class for the variable definitions. The technique has been described earlier for the VAR module, see Section 4.6.3. The STR module also reuses `PCall` classes for several diagnostic tests that are also employed in all other analysis modules. Internal reuse with the help of the `PCall` system makes life easier here as well.

4.10 Nonparametric Analysis

4.10.1 Overview

The nonparametric analysis module in JMulTi allows to specify, estimate, and analyse nonparametric time series models and use them for forecasting. It covers exclusively models for univariate stochastic processes $\{y_t\}_{t \geq 1}$. In the basic nonparametric heteroskedastic nonlinear autoregressive (NAR) model it is assumed that the stochastic process $\{y_t\}$ is generated by the process

$$y_t = \mu(x_t) + \sigma(x_t)\xi_t \quad (4.11)$$

where $x_t = (y_{t-i_1}, y_{t-i_2}, \dots, y_{t-i_m})'$ is the vector of all m correct lagged values, $i_1 < i_2 < \dots < i_m$, and the ξ_t 's, $t = i_m + 1, i_m + 2, \dots$, denote a sequence of i.i.d. random variables with zero mean and unit variance. The functions $\mu(x_t)$ and $\sigma(x_t)$

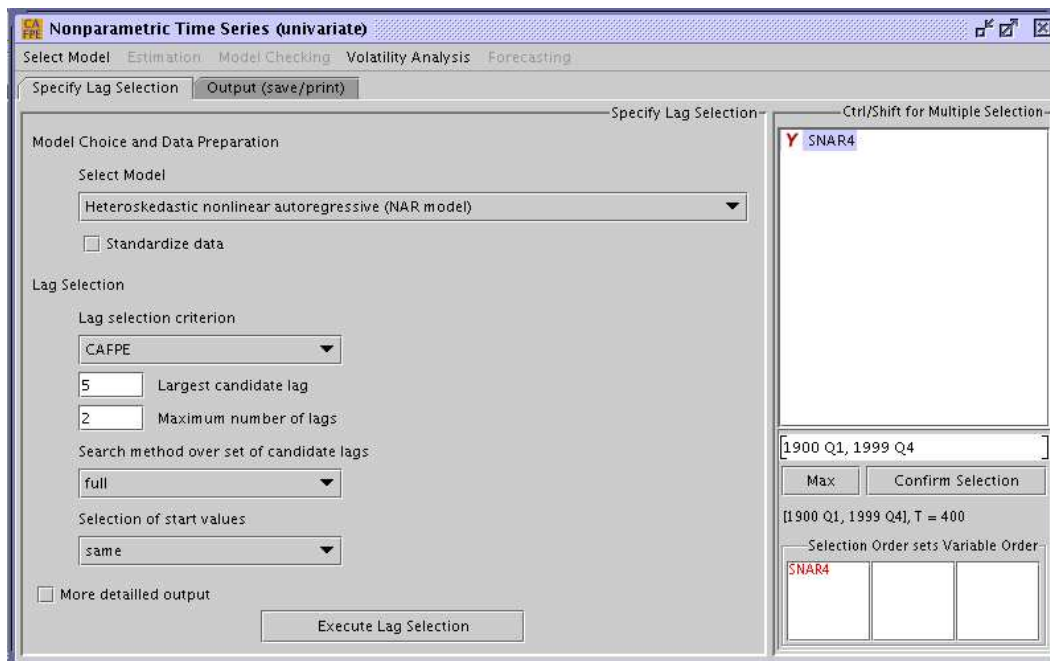


Figure 4.37: Screenshot of model selection for the nonparametric analysis

denote the conditional mean and volatility function, respectively. JMULTi also allows the conditional volatility function $\sigma(\cdot)$ to have lags different from those of the conditional mean function.

In addition to the NAR model there are three seasonal extensions: the seasonal nonlinear autoregressive (SNAR) model, the seasonal dummy nonlinear autoregressive (SDNAR) model, and the seasonal shift nonlinear autoregressive (SHNAR) model. For a comprehensive description of the underlying models, see Tschernig (2004).

A special feature of this module is the automatic selection of all relevant lags for the chosen model according to a specific criterion and a maximum number of lags. This is especially helpful for nonparametric modelling to reduce the impact of the “curse of dimensionality”, which means that the rate of convergence of the estimates decreases with the number of explanatory variables in the model. For this reason it is important to select parsimonious models.

During the modelling process and for model evaluation, it can be useful to additionally employ linear lag selection methods. This can be done in JMULTi for the

linear counterparts of the above mentioned models. These include: the autoregressive (AR) model, the periodic autoregressive (PAR) model, the seasonal dummy linear autoregressive (SDAR) model, and the seasonal shift linear autoregressive (SHAR) model.

Before beginning a nonparametric analysis, one should be aware that all nonparametric methods that are implemented in JMulTi are developed for stochastic processes that are stationary and show a quick enough decay of stochastic dependence between observations with an increasing time span.

The basic modelling steps implied by this analysis module are

1. **Prior Data Transformations** - The first step in nonparametric modelling is to transform the time series such that it becomes stationary and β -mixing. The latter implies that the stochastic dependence between observations of different time points decreases fast enough with the distance between observations. This can be done with standard tools provided by JStatCom, like the Time Series Calculator (see Section 3.24) and the TSSe1 component.
2. **Model and Lag Selection** - An endogenous variable can be selected and automatic selection procedures can be applied to find the relevant lags.
3. **Estimation** - If a nonparametric model was specified before, it is possible to estimate the conditional mean of the process at each time t with the local linear estimator.
4. **Model Checking** - It is possible to check the model against misspecification like in the VAR and VEC modules with a number of diagnostic tests and other tools.

and potentially

5. **Volatility Analysis** - it is possible to specify and estimate the process for the conditional volatility after the conditional mean was estimated. Model checking tools are provided as well.
6. **Forecasting** - One can compute one-step ahead forecasts and conduct rolling over predictions.

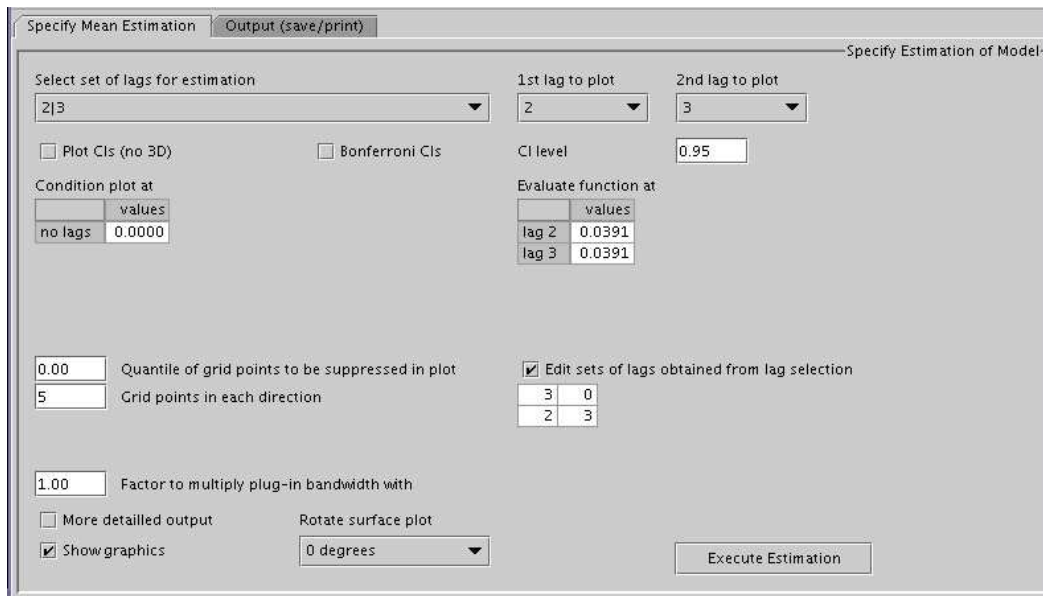


Figure 4.38: Screenshot of model estimation for the nonparametric analysis

4.10.2 Implemented Features

This analysis module is geared towards nonparametric time series modelling, although it can also be used to specify linear models with the lag selection based on the standard information criteria AIC, SC, HQ, and FPE. When linear models are specified, it is not possible to use the estimation panel. One could switch to the VAR analysis module instead to estimate a model with the selected lags. However, the linear models can still be used for forecasting. The nonparametric volatility analysis is available for them as well.

Model Selection

The starting point for the analysis is to select a single stationary variable with the already well known time series selector, see Figure 4.37. Afterwards one can choose an appropriate nonparametric or linear model for which the lag selection should be carried out. If the selected series has quarterly or monthly periodicity then seasonal models are available for selection, otherwise not.

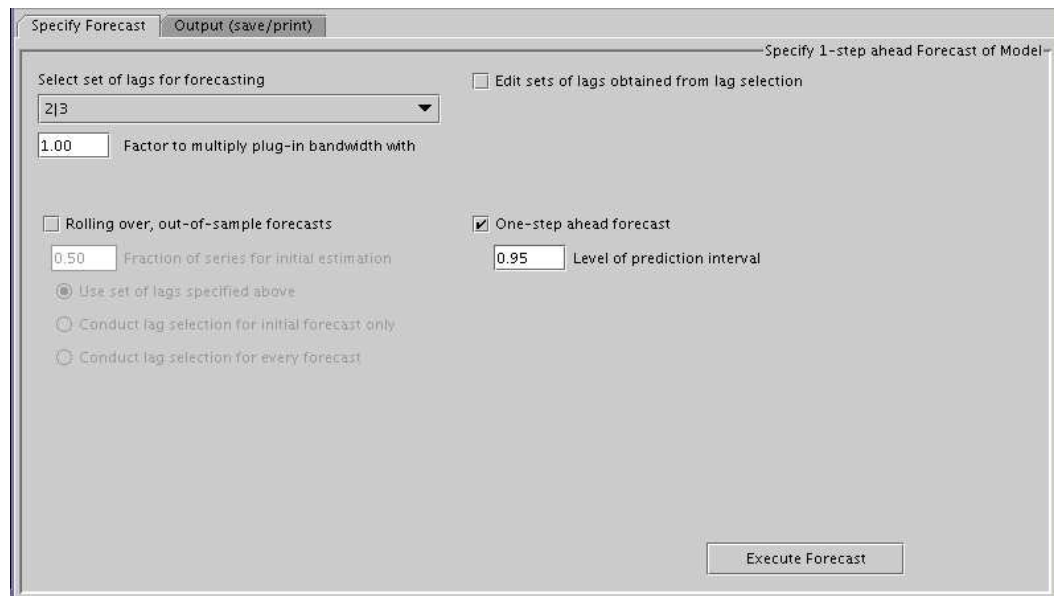


Figure 4.39: Screenshot of forecasts for the nonparametric analysis

The lag selection evaluates the selected criterion for all possible models that have a fixed number of lags and a largest possible lag. The criterion, the maximum number of included lags, as well as the largest candidate lag can be adjusted. The model with the lowest criterion value is suggested to the user.

Available criteria for the lag selection of nonparametric models are the Asymptotic Final Prediction Error (AFPE) and the Corrected Asymptotic Final Prediction Error (CAFPE). The CAFPE is a correction to improve the finite sample behaviour of the AFPE and is less vulnerable to overfitting (Tschernig and Yang (2000)). If a linear model is selected then the already mentioned standard information criteria can be used.

Estimation

If a nonparametric model was selected (NAR, SNAR, SDNAR, or SHNAR) and the lags are already specified, local linear estimation of the conditional mean can be carried out with the estimation panel, see Figure 4.38. It is possible to select the desired lag structure and to evaluate the model at a grid and at a selected point. The nonparametric estimation is typically presented as a plot which is

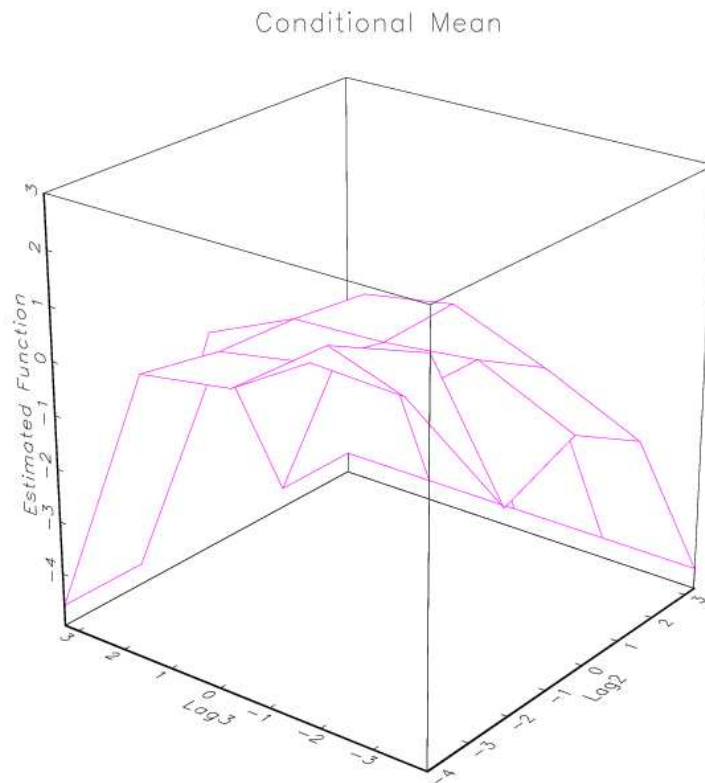


Figure 4.40: Screenshot of surface plot for conditional mean

three-dimensional if there are two free lags, see Figure 4.40. When the selected model includes more lags, then fixed values have to be chosen for the remaining lags. The user can select which lags should be selected for the axes of the graph. It is also possible to plot confidence intervals if there is only one free lag, as shown in Figure 4.41. In this case all other lags, if available, have to be set to fixed values.

Further options of the estimation panel include editing of the selected lags to estimate other models, choosing the kernel bandwidth, adjusting the number of grid points for each lag, as well as miscellaneous plot options.

Model Checking

To check the assumptions about the residuals of the model, similar tools that have already been presented in the VAR and VEC analysis modules can be applied. It is

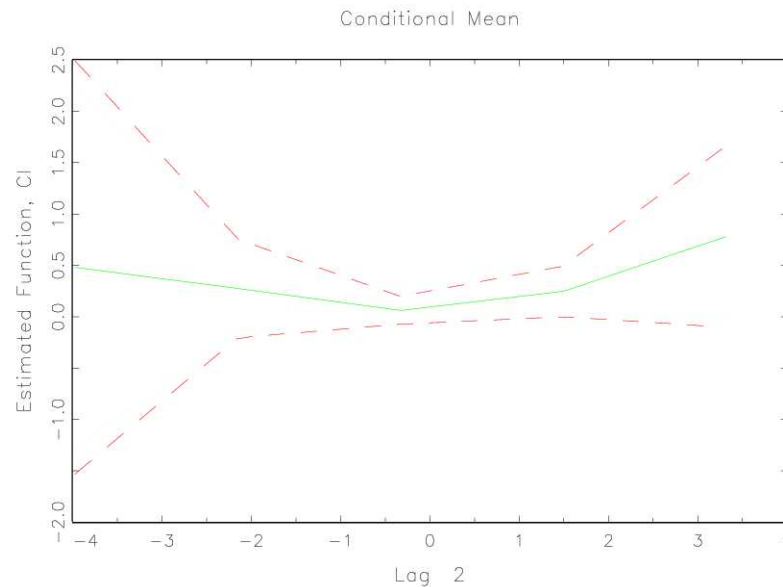


Figure 4.41: Screenshot of conditional mean together with Bonferroni CIs

possible to plot the residuals, to compute the AC and PAC functions, as well as to display the spectrum and kernel density estimates. The Portmanteau, ARCH-LM, Jarque-Bera, and Godfrey (1988) diagnostic tests are also available.

Forecasting

After a linear or nonparametric model has been specified, the forecast panel is activated as soon as the optimal lag structure has been found. With the help of this panel the user can either conduct rolling over, out-of-sample forecasts, or one-step ahead forecasts.

Rolling-over, out-of-sample forecasts For these forecasts one splits the data set into a sample for estimation which contains the first T' values and a sample of the remaining $T - T'$ values for out-of-sample forecasting. The first forecast is computed for $y_{T'+1}$ based on all observations available up to time T' . In the next step one forecasts $y_{T'+2}$ based on the sample $\{y_1, \dots, y_{T'+1}\}$. This procedure is iterated until $T - 1$ observations are used and y_T is forecasted. One then computes

the rolling over one-step ahead prediction error

$$PE = \frac{1}{T - T'} \sum_{j=T'+1}^T (\widehat{y}_j - y_j)^2. \quad (4.12)$$

The use of rolling over predictions is to allow out-of-sample comparisons of the model with other models. For example, one may compare the out-of-sample performance of a NAR model with its linear AR counterpart. Therefore, rolling over predictions can also be conducted for all linear models. Rolling over, out-of-sample predictions can also be computed for all seasonal models: SNAR, SDNAR and SHNAR.

One-step ahead forecast The nonparametric one-period ahead forecast for a time series is obtained by estimating $f(x_{T+1})$ where this estimate is computed using the settings specified in the lag selection panel and the estimation panel respectively. The prediction interval is stated as well in the output window.

Conditional Volatility Analysis

It is possible to model the conditional volatility $\sigma(\cdot)$ nonparametrically after the conditional mean has been estimated. For this, the NAR and SDNAR models are available with the lag selection criteria AFPE and CAFPE. Lag selection for the conditional variance function is required, because in practice the lags in $\sigma^2(\cdot)$ may differ from those in $\mu(\cdot)$. The selected model for $\sigma(\cdot)$ can then be estimated in a similar way than the model for the conditional mean with the respective panel. The residual analysis is activated after the estimation and it has the same features as the model checking panel for the residuals from the estimation of $\mu(\cdot)$. Note that if the conditional mean function and the conditional volatility function of the NAR model are correctly specified, the estimated ξ_t 's should look like an *i.i.d.* sequence with mean 0 and variance 1.

It should be mentioned that the volatility analysis panel is also available before the conditional mean has been estimated. But in this case lag selection can only be done with the option that $\mu(\cdot) = 0$.

4.10.3 Implementation Details

The implementation of the nonparametric analysis module is based on a design that is similar to the other modules. Again, a TSSel component is applied to select the variable, and the model is represented by a set of symbols that are globally shared in the symbol table for the model frame. The variable definitions are declared in an extra class CAFPE_constants because there are 43 shared symbols that are required by different panels.

A challenge for this module was to adjust the set of available models according to the variable selection and to enable/disable the panels for subsequent steps. Seasonal models are available only for data with quarterly and monthly frequencies. For other frequencies these options should not be displayed. The following code shows how this was accomplished:

```
public class CAFPEsel extends ModelPanel{
    private JComboBox modelBox = null;
    // array with all model names
    private static final String[] modelKeys = new String[]{
        "Heteroskedastic nonlinear autoregressive (NAR model)",
        "Seasonal nonlinear autoregressive (SNAR) model",
        "Seasonal dummy nonlinear autoregressive (SDNAR) model",
        "Seasonal shift nonlinear autoregressive (SHNAR) model",
        "Linear autoregressive (AR) model",
        "Periodic autoregressive (PAR) model",
        "Seasonal dummy linear autoregressive (SDAR) model",
        "Seasonal shift linear autoregressive (SHAR) model",
        "Seasonal nonlinear autoregressive (SNAR12) model",
        "Seasonal shift nonlinear autoregressive (SHNAR12) model",
        "Periodic autoregressive (PAR12) model",
        "Seasonal shift linear autoregressive (SHAR12) model",};
    // gets model selection combobox
    private JComboBox getModel() {
        if (modelBox == null)
            modelBox = new javax.swing.JComboBox();
        return modelBox;
    }
}
```

```
// to be called when this class is created
private void initialize(){
    ....
    global().get(CAFPE_constants.DRANGE).addSymbolListener(
        new SymbolListener() {
            public void valueChanged(SymbolEvent evt) {
                getModel().removeAllItems();
                getModel().addItem(modelKeys[0]);
                getModel().addItem(modelKeys[4]);
                if (evt.getSource().isEmpty())
                    return;
                int period = evt.getSource().getJSCDRange()
                    .getTSDateRange().subPeriodicity();
                if (period == 4) {
                    getModel().addItem(modelKeys[1]);
                    getModel().addItem(modelKeys[2]);
                    getModel().addItem(modelKeys[3]);
                    getModel().addItem(modelKeys[5]);
                    getModel().addItem(modelKeys[6]);
                    getModel().addItem(modelKeys[7]);
                }
                if (period == 12) {
                    getModel().addItem(modelKeys[8]);
                    getModel().addItem(modelKeys[9]);
                    getModel().addItem(modelKeys[10]);
                    getModel().addItem(modelKeys[11]);
                }
            }
        });
}
```

The example presents parts of the class `CAFPESel` that is used as the panel for model specification. From the declaration part of the class it can be seen that a `String` array `modelKeys` with the names of all models is created. The method `getModel` initializes a combobox component that holds all options and

can be used to select a model, see Figure 4.37. In the `initialize` method of this class a `SymbolListener` is installed that listens to changes in the selected sample range symbol `DRANGE`. This variable has been declared in the mentioned class `CAFPE_constants` and it holds all information about the selected sample range, in particular the periodicity of the selected series. On a change in that variable, the listener removes all elements from the combobox and first adds the model options that do not depend on the selected periodicity, which are NAR and AR. Then it retrieves the frequency of the series. If this is 4 or 12, corresponding to quarterly or monthly data, it adds the seasonal models that could be used in these cases. This way the user is only presented with valid options.

However, if components adjust their behaviour dynamically this has always the potential for confusion, because the underlying algorithm might not be obvious to the user. Therefore it should at least be documented in the help system. An alternative would have been to keep all possible options and to present a message to the user if an inappropriate model was chosen, which is sometimes a less elegant but more comprehensible solution to similar problems.

Like in the ARCH analysis module, the standard `javax.swing.JComboBox` component, which is not part of `JStatCom`, has been used frequently to let the user select one element from a relatively small set of options. The Data Table system has been applied in the estimation panel (Figure 4.38) to let the user edit the values of lags that should be fixed for the estimation. The dimensions of the presented matrices change according to the selected lag structure. This interaction was also developed with the help of listeners.

A special implementation detail of this analysis module is that the text output from the procedures has not been generated in Java, but has been assembled in the Gauss procedures. The GUI just uses the name of a temporary file as an input parameter for the underlying Gauss procedure and reads the contents of that file after the call has finished. The contents of the output file are then set to the instance of `ResultField`. Another feature is that the output text fields are shown in an extra tab of a tabbed panel because the generated text is quite long. How this was implemented is shown in the code snippet:

```
private OutHolder resultField = null;
private JTabbedPane tabbedPane = null;
...
private void execute(){
    final File outFile = new File(JSCConstants.getSystemTemp()
        + "/cafpe_sel_out");

    // creates instance of lag selection call
    PCall job = ... // takes outFile as argument for procedure

    // set output text field
    job.setOutHolder(resultField);
    // add listener
    job.addPCallListener(new PCallAdapter() {
        public void success() {
            tabbedPane.setSelectedIndex(1);
        }
        public void finished(PCall pCall) {
            outFile.delete();
        }
    });
    job.execute();
}
```

This code is part of the class for the selection panel. The `execute` method is called when the respective button is clicked. A `ResultField` and a `JTabbedPane` are part of this panel, the initialization methods are omitted here. In the `execute` method first the file name `outFile` is generated with the help of the `FileSupport` class. The method `getSystemTemp` gets the name of the temporary directory to which all applications have write access. Then a caller object is created which must take `outFile` as an argument, details are again left out. The `job` object is then told which component to use for printing the results, which is `resultField` in that case. How the contents of the output file are read back and set to the text component after the computation is over is also handled by the `job` object.

Finally, the calling panel installs a listener to the caller object to take some action when the call finishes. For this the `PDataAdapter` class is subclassed, which has been described in Section 3.19. The listener sets the selected tab to the output field if the call finished successfully, see Figure 4.42. Therefore it overwrites the method `success`. In any case it deletes the output file by overwriting `finished`. Finally, the caller job is executed which means that it is queued for execution in the background thread.

```

Specify Lag Selection  Output (save/print)
Lag vector considered:      3  5
Lag vector considered:      4  5

Results:
-----

I. Main results:

Selected lags/indices of variables using arfpe:
  1
Associated criterion value:                1.4238623
Estimated variance of white noise process: 1.4059295

II. Detailed results:

No. of lags      selected vector of lags/indices
  1                1  0
  2                1  4

No. of lags estim.  crit. value      est. var of WN  est. value of A (if (C)AFPE)
  0                1.4937676      1.4937676      1.4937676
  1                1.4238623      1.4059295      1.4059295
  2                1.4297348      1.408181       1.408181
CAPFE program terminated.

```

Figure 4.42: Screenshot of text output after lag selection finished

4.11 Outlook

The software JMulTi is an ongoing project that is now part of the newly created National Research Center 649 “Economic Risk” at the Humboldt-Universität zu Berlin in the project “Unit Root and Cointegration Methods” (C2). JStatCom will serve as a platform to make the developed econometric methods available to other researchers.

A general limitation of the software JMulTi is currently the lack of project management. Therefore it is not possible to save the state of a model to a file, but it is necessary to reproduce all specification steps when a certain model should be restored. However, this is a feature that is being worked on and that can be programmed on top of the existing implementation without the need for a general redesign.

Another limitation is that the software uses the Gauss graphics engine which does not allow to edit graphs that have been created. It would also be desirable to store created graphs for a model as part of a project. As already mentioned, it is likely that these features can be implemented when a Java graphics engine is used.

Of course there are also other econometric features that should be integrated in the JMulTi software. For example, a module especially designed for univariate time series analysis would be what many users would expect in such a software.

Further extensions include better support for importing data from various sources. It is currently planned to make it possible to read in Gauss `.fmt` and `.dat` files. Data export to these file formats as well as to Excel workfiles is also being worked on. These features can be implemented on the framework level and do not conflict with its current design.

A general problem is always the quality of the used algorithms in terms of stability and performance. Experience shows that most errors occur in the algorithms. This situation can be improved by consequently applying unit testing on the caller classes. For this it is especially useful that test data can be stored in text files which can easily be read into instances of `JSCNArray`. This makes it possible to use real data for unit testing and to check the results of the econometric procedures via assertions.

4.12 Conclusion

It has been mentioned throughout the text that JStatCom provides standard solutions to recurring tasks. The descriptions of the modules should have underlined this statement by showing how problems occurring in quite different theoretical models could be solved in similar ways on the basis of systems provided by JStatCom. It is argued that this is a big advantage over other approaches, because

it allows to implement a wide range of models with similar programming techniques. Developers can therefore focus more on the algorithms, on the quality of the presented GUI solutions, and on documentation, instead of inventing new solutions for data import, selection, help system integration, and data representation for every model they want to supply with a graphical user interface.

The chosen approach has successfully been used for the implementation of several analysis modules with quite different behaviour. From these examples one can see that it is general enough to be used for any other model in time series analysis. Furthermore, the framework can also be used in other problem domains where similar tasks have to be solved. The extension mechanisms provided by JStatCom might be used to develop the required adjustments. For example, the TSSel component is especially designed to select time series and might not be applicable in other domains where the data has different properties.

It is therefore hoped that the software framework JStatCom will be used to implement various tools to support empirical analysis by making up-to-date methods available to the practitioner and to other researchers. The framework will also be used to extend the functionality of existing modules in JMulTi and to develop new modules for other models, thus making the software an even better candidate for classroom use and research.

Appendix A

Guide to Notation

The description of the framework uses formal and informal graphical representations to illustrate important aspects of the system. Informal diagrams are screenshots that should give an intuitive idea about the operation of the discussed part of the system. Formal diagrams use the UML modelling language (Booch et al. (1999)) to describe classes, objects, and their relationships.¹ They are also used to show the context of a system in relation to its environment. Other representations include tables to describe the elements of a system together with a short description of their responsibilities. Those tables are formal in a way that they list all elements that have an *is-part-of* relationship to the subsystem under consideration, and they are informal in the way responsibilities are described.

It is important to note that the notation should capture only selected aspects that are relevant to understand the architecture. Therefore, the chosen UML diagrams often do not display all classes that are involved, and they typically show only certain methods and fields, if any. Otherwise, the use of these diagrams would be quite limited because they would simply get too big. Moreover, detailed and complete information about all classes is given in the API documentation.

This architecture documentation does include only a small subset of all available diagram types in the UML. The focus is mostly on the static class structures, and it is argued that this is the most relevant type of information for developers using JStatCom. However, information about behaviour and the allocation of related

¹All diagrams in this thesis have been generated with the UML modelling software *Poseidon for UML 2.0, Community Edition*. The URL for the software is www.gentleware.com.

resources is given in the textual description of some subsystems when necessary. The following UML diagram types have been used:

- **class diagram** - Depicts classes, their structure, and relationships between them. Elements are classes and interfaces. In some cases, also components and instances of components are elements of class diagrams.
- **object diagram** - Presents a particular object structure at runtime. An object is an instance of a class.
- **use case diagram** - Shows a set of use cases and actors and their relationships. It depicts how a system interacts with its environment.

In the remaining sections, all elements of the used diagrams are described.

A.1 Class Diagrams

A.1.1 Elements and Inheritance

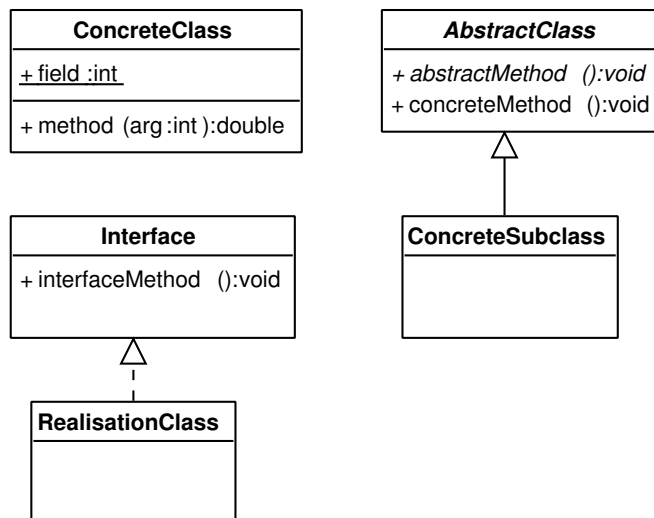


Figure A.1: Classes and inheritance

Figure A.1 shows different elements of a class diagram and the use of the *inherits* and *realizes* relations. The class `ConcreteClass` has an interface together with an implementation. It is neither an interface nor an abstract class and objects can be created directly by invoking its constructor. A class can have fields and methods. Fields are shown in the first compartment, and methods are shown with their signature in the second compartment. Mostly, only methods are presented, because direct access to fields is a special case. Therefore, the first compartment is often not shown at all.

- **field** - A '+' denotes that the field is publicly visible, a '-' denotes private fields that can only be accessed from inside the class. The name of the field is given first, followed by a ':' and the type. An underline marks that the variable is static. This means that it is shared by all instances of the class it belongs to, as opposed to nonstatic fields which belong to an object.
- **method** - A '+' denotes that the method is publicly visible, a '-' denotes private methods that can only be accessed from inside the class. The name of the method is given first, followed by the arguments with type and name in brackets. After the ':', the return type is given. An underline marks that the method is static. This means that it can be invoked without creating an instance of the class it belongs to. Nonstatic method can only be invoked on objects.

As opposed to concrete classes, there are also **abstract classes**. One cannot create instances of abstract classes directly. They have to be subclassed first and all methods declared as `abstract` have to be implemented. Abstract classes are useful to provide default functionality that is shared by all implementing classes and that can be factored out in a single class. At the same time, subclassing is enforced to implement the varying parts. Typically, clients only need to interact with the interface of an abstract class. `AbstractClass` is an example of such a class. It has the abstract method `abstractMethod` and the concrete method `concreteMethod`. Abstract classes and methods are indicated by the italicised font.

Because abstract classes cannot be instantiated, they have to be subclassed. The `ConcreteSubclass` is therefore an example of a class that inherits from `AbstractClass`. It must provide an implementation for `abstractMethod`. The *inherits* relation is shown by the arrow with a solid line.

It should be mentioned that this relation would also be possible between non abstract classes and subclasses. However, because of the problems that inheritance can cause, subclassing is typically allowed only for abstract classes, except in special cases. Otherwise it is programmatically prohibited via the `final` keyword. However, this is not visible from UML class diagrams. For a justification of this relatively strict policy, see Bloch (2001), item 15.

Another very important element of class diagrams are **interfaces**. An interface is nothing more but a set of methods that classes promise to implement in a certain way. Interfaces never contain any concrete methods. Classes that implement an interface are said to be **realisations** of it. The *realises* relationship is denoted as an arrow with a dotted line. `Interface` and `RealisationClass` of Figure A.1 are an example for this relation. Despite its conceptual simplicity, interfaces are one of the most powerful constructs of the Java language. They are heavily used throughout JStatCom to decouple interface definitions from their actual implementations.

A.1.2 Components



Figure A.2: Components

Figure A.2 shows how components and instances of components can be represented with the UML. In the JStatCom architecture documentation those elements are occasionally shown together with classes or objects. The notion of components is used to represent a process, a dynamic link library, or a file with

configuration information or algorithms. These are either represented as a component, in case of a file on the disk, or as an instance of a component, for example if a process is created from an executable file. Classes can have a dependency relation to components.

A.1.3 Relations between Elements of Class Diagrams

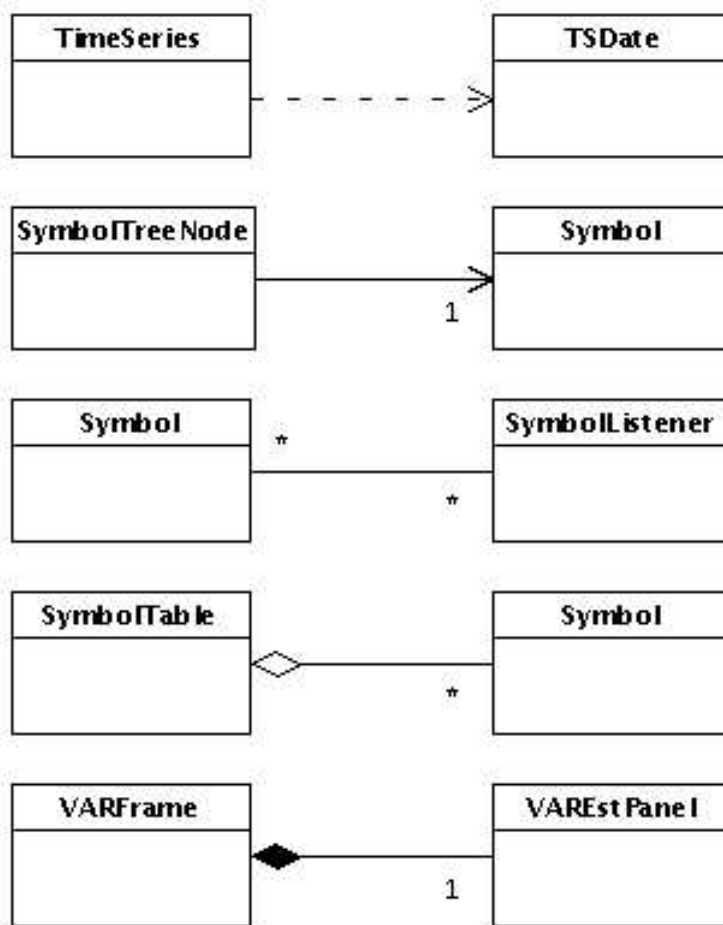


Figure A.3: Relations among classes

Figure A.3 presents relations between classes that are not related to inheritance. They are named and described in the following:

- **dependency** (pair 1) - Indicates that one class uses another class. This means that the behaviour of the dependent class is affected by changes in the independent class, but not vice versa. Typically, this means that the other class is used in a method or is part of a method signature of the dependent class. `TimeSeries` depends on `TSDate` because the date is part of the constructor signature. It is used to determine the sample range. `TSDate` does not depend on `TimeSeries` in any way.
- **association** (pairs 2, 3) - Structural relationship that describes a set of links between instances of the involved classes. Typically, it means that instances of a class keep zero, one, or many references to instances of another class. The multiplicity is indicated with:
 - '*' - any number, including zero
 - '1..*' - any number, but at least one
 - '1' - exactly one
 - '0..1' - zero or one

The association can be directed or undirected. A direction is indicated with an arrow. A `SymbolTreeNode` has exactly one reference to a `Symbol`, but the symbol does not know anything about the tree node referencing it. This is a directed association. An example of a bidirectional link is pair 3. Each `Symbol` can have an arbitrary number of `SymbolListener` instances, whereas each listener can register itself to an arbitrary number of symbols as well.

- **aggregation** (pair 4) - A special form of association that defines a *whole-part* relationship between the aggregate and the part. It is indicated by an empty diamond on the side of the aggregate. Because it is a specialization of an association, the same rules for multiplicity apply. A `SymbolTable` is an aggregation of zero, one, or many `Symbol` instances. To mark the

difference to a composition, one has to note that a symbol object can also exist without the symbol table object that aggregates it.

- **composition** (pair 5) - A form of aggregation with strong ownership of the parts by the whole. The aggregate is also called composite. The lifetime of the parts finishes when the composite dies. The same rules for multiplicity apply as for an association. A VARFrame has exactly one instance of a VAREstPanel, which has no life outside the VAR frame. This means that no other objects that are independent of the VAR frame keep references to the estimation panel. GUI components are typical examples of composites.

A.2 Object Diagrams

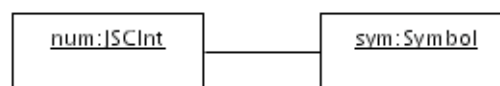


Figure A.4: Object diagram

Object diagrams capture the runtime structure of instances of classes at a point in time for real or prototypical cases. Figure A.4 shows two objects that are linked. An instance of Symbol refers to a JSCInt data object.

As opposed to class diagrams, an object diagram might contain more than one instance of a class appearing, whereas it would not make sense to have a class appearing more than one time in a class diagram. Every object is defined by its name and its type, which are separated by a ':'. The relation between objects is a link, which is an instance of an association.

A.3 Use Case Diagrams

Use case diagrams capture how a certain system is related to its environment. Figure A.5 shows an example. The elements are actors, use cases and packages. An actor is a person or another system that interacts with the system under consideration. The latter is often represented by a rectangular area, which is called package.

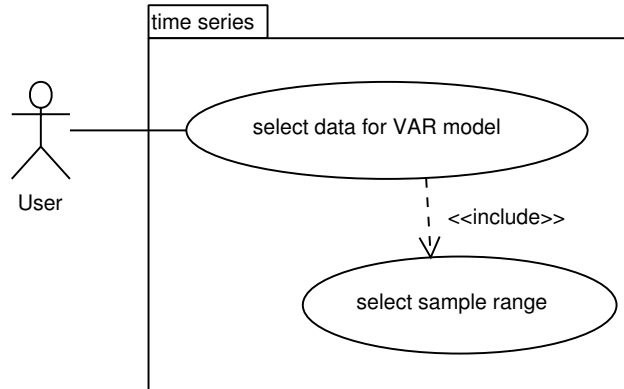


Figure A.5: Use case diagram

Use cases are certain behavioural elements that are invoked by the actor. A use case can include other sub tasks that belong to it. Those subtasks might also be included by other use cases as well. For example, selecting the variables for a VAR model would include the task for selecting the sample range. For the JStat-Com architecture documentation, use cases are employed to show the context of a certain system, as suggested in Clements et al. (2003).

Appendix B

Documenting Modules with JavaHelp and JHelpDev

"The JavaHelp system provides developers and authors a standard, fully-featured, easy-to-use system for presenting online information to Java application users. Providing a help system standard that is part of the Java platform relieves developers and authors of the task of inventing their own proprietary help systems."
(JHelp User Guide)

Modern applications that claim to be professionally developed have to provide users with a state-of-the-art help system. It must integrate well with the underlying application, must be easy to browse and search, and should also be context-sensitive. For Java, there exists the JavaHelp system which gives developers the needed functionality. It is by now the de facto standard for help systems in Java applications and replaces ad hoc solutions or simple PDF files.

The problem domain of data based analysis, especially empirical econometrics, requires yet another special feature of a help system. It must be possible for authors to write documents with many mathematical formulas conveniently, and to convert them easily to the required help format. Like most other help systems, the JavaHelp system is based on HTML, meaning that the content has to be written in that language.

It is clear that this is obviously not an optimal solution for coding mathematical formulas, because they either have to be written in a textual form, or they have to be converted to a graphics file that can be embedded in an HTML file. The first approach simply does not look very well, the second solution depends very much on how the graphics creation is handled. Therefore, direct help authoring in HTML is tedious and not very effective.

B.1 LaTeX and latex2html

Because of the above mentioned problems, it is argued that help authoring must be made more convenient to encourage developers or scientists to provide documentation for their modules. This requires a standard approach to creating documents in the problem domain that can be adopted without imposing further learning costs. Most scientists write their texts in LaTeX and are familiar with that system. The suggested solution is therefore to create help documents in LaTeX and use the conversion tool `latex2html` for generating HTML files from it.¹ It has been found that this gives acceptable results. The workings of the `latex2html` tool is not described here, but it is available on all major operating systems.

As an example, assuming that the file `help.tex` holds the main LaTeX file for the helpsystem of a certain module, the following command could be used to initiate the conversion:

```
latex2html -split 4 -link 1 -white -local_icons  
           -notransparent -address " " -noinfo help.tex
```

This would result in a subdirectory with HTML files and graphics. The commandline options are not described here in detail, but can be checked with the `latex2html` documentation.

¹The URL for the software `latex2html` is www.latex2html.org.

B.2 JHelpDev

When the first step is done and the HTML files are there, the actual helpset must be created in the JavaHelp format. A helpset consists of several XML files:

- `Map.jhm` - defines mappings between so called target IDs and relative URLs of the HTML files
- `XXX.hs` - the main configuration file of the helpset, specifies title and other settings for the general appearance
- `XXXIndex.xml` - can be used to create an index with keywords that are linked with target IDs
- `XXXTOC.xml` - defines the table of contents definition, each entry should point to a target ID

`XXX` is a placeholder for the name of the helpset, for example `VAR`, `VEC` or `STR`. It can be chosen freely.

The mentioned XML files can in principle be created by hand, but this is very ineffective and error-prone. Therefore, tool support is needed to create the helpset easily. Although there exist tools for creating JavaHelp sets, some of them freely available, there is need for a special solution that automatically generates the mappings, and possibly the table of contents, from an existing set of HTML files. This was the motivation to develop the software JHelpDev.² Most other help authoring tools focus on editing the HTML content files. JHelpDev does not provide HTML editing capabilities at all, but just indexes files in a given directory tree and offers editors for the table of contents and the help index. Thus, it can be used to create a working JavaHelp set from the latex2html output with only a few mouse clicks. Figure B.1 shows the table of contents editor of JHelpDev.

² The URL for the software JHelpDev is jhelpdev.sourceforge.net. It is published under the LGPL and is hosted as a sourceforge project. The main development was done by M. Krätzig, recently some add-ons have been contributed by the registered developer J. Iry.

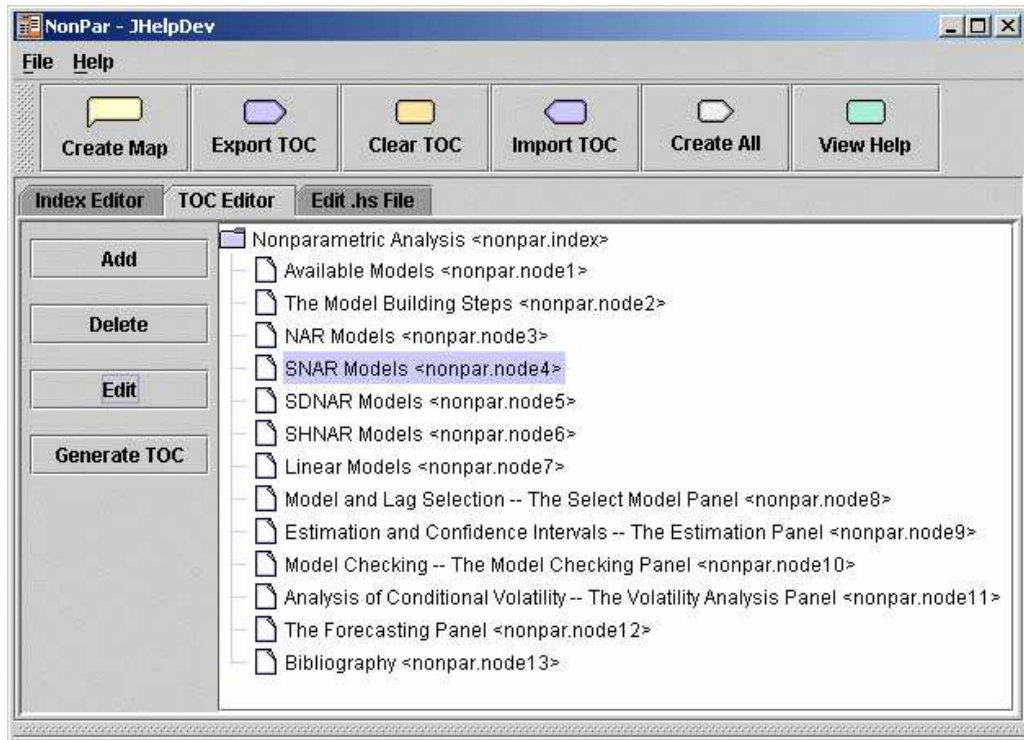


Figure B.1: Screenshot of a TOC editor component

The operation of the software is described in greater detail in the integrated help system. But the general steps are the following:

- create a new project by telling the program which directory holds the HTML files to be used
- once confirmed, JHelpDev automatically creates all required XML files, including a default table of contents
- typically, the table of contents is edited to change the order and to add, delete, or rename items
- an index can optionally be created with the index editor
- by clicking on *Create All*, the helpset is generated with the full text search database

- the final result can always be checked via the *View Help* button

The whole project directory should then be copied into the resource directory of the application, typically in a subdirectory `helpsets/XXX`. A screenshot of a helpset that has been generated this way is shown in Figure B.2. It is actually part of the JMulTi application.

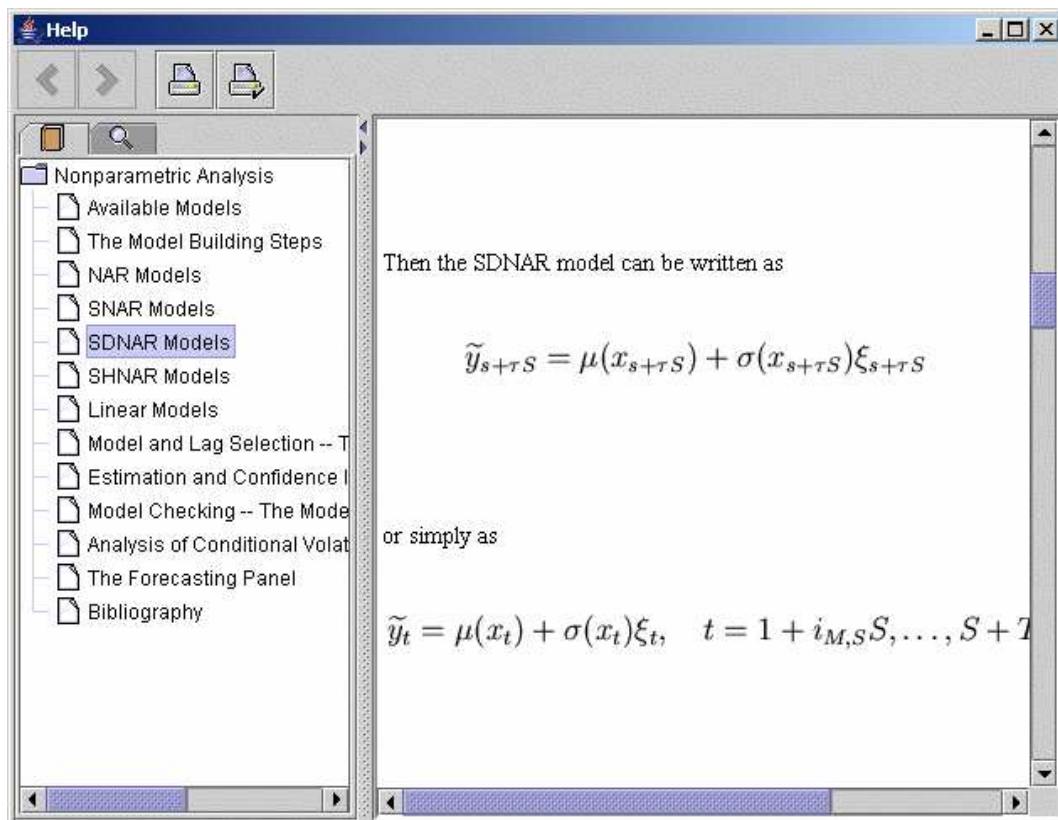


Figure B.2: Screenshot of a help viewer

B.3 Integrating Helpsets with an Application

The standard way of integrating a helpset is to put it in the file `modules.xml`, as described in the Application view package, see Section 3.26. The filename must point to the relative location of the main configuration file `XXX.hs`. When the

application is started via the `TopFrame` class, all helpsets are created and dynamically merged. Dynamic merging is a feature provided by the JavaHelp system. It just results in a table of contents that contains the elements of all merged helpsets together.

For the target IDs that are needed for setting up the context-sensitive help via the file `help_context.xml`, one should check the TOC editor in the JHelpDev tool. For each element the target ID is shown as well. It is also possible to read the `Map.jhm` or `XXXTOC.xml` files directly and to retrieve the IDs from there.

However, there is just one little problem. When creating context-sensitive help, components are mapped to target IDs. A target is always unique when a helpset is created, but it might not be unique anymore when several helpsets are merged that use the same target IDs. In this case, the context-sensitive help might point to the wrong topic. To avoid this, one has to note that JHelpDev generates target IDs from the relative filenames of the HTML files. Therefore one should pay attention to make these paths contain a descriptive directory name that is unique for every helpset. For example, all HTML files for the VAR help could go into a directory `help/varhelp`. The JHelpDev project directory might then be `help`. In this case, all target IDs for the generated helpset would contain the string `varhelp`, for example `varhelp.node1`, which can distinguish it from targets of other helpsets. The problem is quite likely to occur, because the HTML files have standardized names when they are created with `latex2html`. Putting them in a subdirectory with a descriptive name will make the IDs unique.

Bibliography

- Akaike, H. (1969). Fitting autoregressive models for prediction. *Annals of the Institute of Statistical Mathematics*, 21:243–247.
- Akaike, H. (1973). Information theory and an extension of the maximum likelihood principle. In Petrov, B. and Csáki, F., editors, *2nd International Symposium on Information Theory*, pages 267–281, Budapest. Akadémiai Kiadó.
- Amisano, G. and C. Giannini (1997). *Topics in Structural VAR Econometrics*. Springer, Berlin, 2nd edition.
- Anderson, E. and Bai, Z. and Bischof, C. and Blackford, S. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.
- Ashworth, M. and Allan, R.J. and Müller, C.J. and van Dam, H.J.J. and Smith, W. and Hanlon, D. and Searly, B.G. and Sunderland, A.G. (2003). Graphical user environments for scientific computing. Technical report, Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Warrington.
- Baba, Y. and Engle, R.F. and Kraft, D.F. and Kroner, K.F. (1990). Multivariate simultaneous generalized ARCH. mimeo, UCSD.
- Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition.
- Benkwitz, A. (2002). *The Software JMulTi: Concept, Development and Application in VAR Analysis*. Dissertation, Humboldt-Universität zu Berlin.

- Berard, Edward V. (1993). Object Class Specification. In *Essays on Object-Oriented Software Engineering*, pages 131–162. Simon & Schuster, Englewood Cliffs, NJ.
- Bianchi, A. and Caivano, D. and Lanubile, F. and Visaggio, G. (2001). Evaluating Software Degradation through Entropy. In *Proc. 7th IEEE International Software Metrics Symposium*, pages 210–219, London.
- Bloch, J. (2001). *Effective Java*. Addison-Wesley.
- Boisvert, R.F. and Dongarra, J.J. and Pozo, R. and Remington, K.A. and Stewart, G.W. (1998). Developing numerical libraries in Java. *Concurrency: Practice and Experience*, 10(11-13):1117–1129.
- Ronald F. Boisvert and José Moreira and Michael Philippsen and Roldan Pozo (2001). Numerical Computing in Java. *Computing in Science and Engineering*, 3(2):18–24.
- Boisvert, R. F. and Tang, P. T. P., editors (2001). *The Architecture of Scientific Software, IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software, October 2-4, 2000, Ottawa, Canada*, volume 188 of *IFIP Conference Proceedings*. Kluwer.
- Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31:307–327.
- Booch, G. and Rumbaugh, J. and Jacobsen, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.
- Breitung, J. and Brüggemann, R. and Lütkepohl, H. (2004). Structural Vector Autoregressive Modelling and Impulse Responses. In Lütkepohl, H. and Kräzig, M., editors, *Applied Time Series Econometrics*. Cambridge University Press.
- Brown, R. L. and Durbin, J. and Evans, J. M. (1975). Techniques for testing the constancy of regression relationships over time. *Journal of the Royal Statistical Society B*, 37:149–192.

- Brüggemann, R. and Lütkepohl, H. (2001). Lag selection in subset VAR models with an application to a U.S. monetary system. In Friedmann, R., Knüppel, L., and Lütkepohl, H., editors, *Econometric Studies: A Festschrift in Honour of Joachim Frohn*, pages 107–128. LIT Verlag, Münster.
- J. M. Bull and L. A. Smith and L. Pottage and R. Freeman (2001). Benchmarking Java against C and Fortran for scientific applications. In *Java Grande*, pages 97–105.
- Candelon, B. and Lütkepohl, H. (2000). On the reliability of Chow type tests for parameter constancy in multivariate dynamic models. Discussion paper, Humboldt-Universität Berlin.
- Clements, P. and Bachmann, F. and Bass, L. and Garlan, D and Ivers, J. and Little, R. and Robert, N and Stafford, J. (2003). *Documenting software architecture: views and beyond*. Addison-Wesley.
- Cribari-Neto, F. and Jensen, M. J. (1997). MATLAB as an Econometric Programming Environment. *Journal of Applied Econometrics*, 12(6):735–44.
- Cribari-Neto, F and Zarkos, S.G. (1999). R: Yet another econometric programming environment. *Journal of Applied Econometrics*, 14:319–329.
- Cribari-Neto, F. and Zarkos, S. G. (2003). Econometric and Statistical Computing Using Ox. *Computational Economics*, 21:277–295.
- Deitel, H. M. and Deitel, P.J. (2002). *C++ How to Program*. Prentice-Hall, 2nd edition.
- Deutsch, L. P. (1989). Design reuse and frameworks in the Smalltalk-80 system. In Biggerstaff, T. J. and Perlis, A. J., editors, *Software Reusability, Volume II: Applications and Experience*, pages 57–71. Addison-Wesley, Reading, MA.
- Dickey, D. A. and Fuller, W. A. (1979). Estimators for Autoregressive Time Series with a Unit Root. *Journal of the American Statistical Association*, 74:427–431.
- Edsger W. Dijkstra (1969). Structured programming. circulated privately.

- Doolin, D. M. and Dongarra, J. (1997). JLAPACK - Compiling LAPACK FORTRAN to Java.
- Doornik, J.A. (2002a). Object-oriented Programming in Econometrics and Statistics using Ox: A Comparison with C++, Java and C#. In Nielsen, S., editor, *Programming Languages and Systems in Computational Economics and Finance*, pages 115–147. Dordrecht: Kluwer Academic Publishers.
- Jurgen Doornik (2002b). *Ox Appendices*.
- Doornik, J.A. and Ooms, M. (2001). *Introduction to Ox*. Timberlake Consultants Press, London.
- Doornik, J. A. (1998). Approximations to the asymptotic distributions of cointegration tests. *Journal of Economic Surveys*, 12:573–593.
- Doornik, J. A. and Hansen, H. (1994). A practical test of multivariate normality. unpublished paper, Nuffield College.
- Doornik, J. A. and Hendry, D. F. (1997). *Modelling Dynamic Systems Using PcFiml 9.0 for Windows*. International Thomson Business Press, London.
- Eckstein, R. and Lay, M. and Wood, D. (1998). *JAVA Swing*. O'Reilly.
- Eddelbuettel, D. (2000). Econometrics with Octave. *Journal of Applied Econometrics*, 15(5):531–542.
- Edgerton, D. and Shukur, G. (1999). Testing autocorrelation in a system perspective. *Econometric Reviews*, 18:343–386.
- Efron, B. and Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall, New York.
- Engle, R. F. (1982). Autoregressive conditional heteroscedasticity, with estimates of the variance of United Kingdoms inflations. *Econometrica*, 50:987–1007.
- Fowler, M. (1999). *Refactoring*. Addison-Wesley.

- Franses, P. H. (1990). Testing for seasonal unit roots in monthly data. Econometric Institute Report 9032A, Erasmus University Rotterdam.
- Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.
- David Garlan and Robert T. Monroe and David Wile (2000). Acme: Architectural Description of Component-Based Systems. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press.
- Glosten, L. and Jagannathan, R. and Runkle, D. (1993). Relationship between the expected value and the volatility of the nominal excess return on stocks. *Journal of Finance*, 48:1779–1801.
- Günther, O. and Müller, R. and Schmidt, P. and Bhargava, H. and Krishnan, R. (1997). MMM: A WWW-based approach for sharing statistical software modules. *IEEE Internet Computing*, 1(3):59–68.
- Godfrey, L.G. (1988). *Misspecification Tests in Econometrics*. Cambridge University Press, Cambridge.
- Haase, K. and Lütkepohl, H. and Claessen, H. and Moryson, M. and Schneider, W. (1992). *MulTi: A Menu-Driven GAUSS Program for Multiple Time Series Analysis*. Universität Kiel, Kiel, Germany.
- Hall, P. (1992). *The Bootstrap and Edgeworth Expansion*. Springer, New York.
- Halloway, Stuart Dabbs (2002). *Component Development for the Java Platform*. developmentor series. Addison-Wesley.
- Hamilton, J. D. (1989). A new approach to the economic analysis of nonstationary time series and the business cycle. *Econometrica*, 57:357–384.
- Hannan, E. J. and Quinn, B. G. (1979). The determination of the order of an autoregression. *Journal of the Royal Statistical Society*, B41:190–195.

- Hansen, H. and Johansen, S. (1999). Some tests for parameter constancy in cointegrated VAR-models. *Econometrics Journal*, 2:306–333.
- Harvey, A. C. (1990). *The econometric analysis of time series*. Philip Allan, Hemel Hempstead, 2nd edition.
- Herwartz, H. (2004). Conditional Heteroskedasticity. In Lütkepohl, H. and Kräzig, M., editors, *Applied Time Series Econometrics*. Cambridge University Press.
- Härdle, W. and Klinke, S. and Turlach, B.A. (1995). *Xplore: An Interactive Statistical Computing Environment*. Springer-Verlag, Berlin.
- Hutton, J (1995). The Maple Computer Algebra System: A Review. *Journal of Applied Econometrics*, 10(3):329–337.
- Hylleberg, S. and Engle, R. F. and Granger, C. W. J. and Yoo, B. S. (1990). Seasonal Integration and Cointegration. *Journal of Econometrics*, 44:215–238.
- IEEE (2000). *IEEE Std 1471 Recommended Practice for Architectural Description*. IEEE.
- Jarque, C. M. and Bera, A. K. (1987). A test for normality of observations and regression residuals. *International Statistical Review*, 55:163–172.
- Johansen, S. (1995). *Likelihood-based Inference in Cointegrated Vector Autoregressive Models*. Oxford University Press, Oxford.
- Johansen, S. and Mosconi, R. and Nielsen, B. (2000). Cointegration Analysis in the Presence of Structural Breaks in the Deterministic Trend. *Econometrics Journal*, 3:216–249.
- Johnson, R. E. and Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35.
- Kwiatkowski, D. and Phillips, P. C. B. and Schmidt, P. and Shin, Y. (1992). Testing the null of stationarity against the alternative of a unit root: How sure are we that the economic time series have a unit root? *Journal of Econometrics*, 54:159–178.

- Lanne, M. and Lütkepohl, H. and Saikkonen, P. (2002). Comparison of unit root tests for time series with level shifts. *Journal of Time Series Analysis*, 23(6):667–685.
- Lea, Doug (2000). *Concurrent Programming in Java, Design Principles and Patterns*. Addison-Wesley, 2nd edition.
- Liang, Sheng (1999). *Java Native Interface*. Addison-Wesley.
- Liu, L.-M. and Montgomery, A.L. and Chan, K.K. and Muller, M.E. (1995). A System-Independent Graphical User Interface for Statistical Software. *Computational Statistics and Data Analysis*, 19:23–44.
- Ljung, G. M. and Box, G. E. P. (1978). On a measure of lack of fit in time-series models. *Biometrika*, 65:297–303.
- Lundbergh, S. and Teräsvirta, T. (2002). Evaluating GARCH models. *Journal of Econometrics*, 110:417–435.
- Lütkepohl, H. (1991). *Introduction to multiple time series analysis*. Springer Verlag, Berlin.
- Lütkepohl, H. and Kräzig, M., editors (2004). *Applied Time Series Econometrics*. Cambridge University Press, Cambridge.
- Sun Microsystems (1.01-1997). JAVA Beans API Specification.
- Sun Microsystems (2002). Model-View-Controller.
- Sun Microsystems (2003). JavaHelp System.
- Metsker, Steven John (2001). *Building Parsers with Java*. Addison-Wesley.
- Jose E. Moreira and Samuel P. Midkiff and Manish Gupta (2001). Supporting Multidimensional Arrays in Java.
- Oheimb, David von (2001). *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München. <http://www4.in.tum.de/~oheimb/diss/>.

- Pree, W. (1997). Component-Based Software Development - A new Paradigm in Software Engineering? In *Software-Concepts & Tools*. Springer-Verlag, Heidelberg/New York.
- Pree, W. (2000). Hot-Spot-Driven Framework Development. In M. Fayad, D. Schmidt, R. J., editor, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley & Sons, New York.
- Press, W.H. and Teukolsky, S.A. and Vetterling, W.T. and Flannery B.P. (2002). *Numerical Recipes*. Cambridge University Press, 2nd edition.
- Riessen, G. and Jacobsen, H.-A. and Günther, O. (2000). Component leasing on the World Wide Web. *Netnomics*, 2:191–219.
- Rose, C. and Smith, M. (2002). A Review of Mathematical Statistics with Mathematica. *Journal of the American Statistical Association*, 97(460):1202–1203.
- Saikkonen, P. and Lütkepohl, H. (2000). Testing for the cointegrating rank of a VAR process with an intercept. *Econometric Theory*, 16:373–406.
- Saikkonen, P. and Lütkepohl, H. (2002). Testing for a unit root in a time series with a level shift at unknown time. *Econometric Theory*, 18:313–348.
- Schmidt, P. and Phillips, P. C. B. (1992). LM tests for a unit root in the presence of deterministic trends. *Oxford Bulletin of Economics and Statistics*, 54:257–287.
- Schwarz, G. (1978). Estimating the dimension of a model. *Annals of Statistics*, 6:461–464.
- Teräsvirta, T. (1998). Modeling economic relationships with smooth transition regressions. In Ullah, A. and Giles, D., editors, *Handbook of Applied Economic Statistics*, pages 229–246. Dekker, New York.
- Teräsvirta, T. (2004). Smooth Transition Regression Modelling. In Lütkepohl, H. and Krätzig, M., editors, *Applied Time Series Econometrics*. Cambridge University Press, Cambridge.

- Trenkler, C. (2004). Determining p-values for systems cointegration tests with a prior adjustment for deterministic terms. mimeo, Humboldt-Universität zu Berlin.
- Tschernig, R. (2004). Nonparametric Time Series Modeling. In Lütkepohl, H. and Kräzig, M., editors, *Applied Time Series Econometrics*. Cambridge University Press.
- Tschernig, R. and Yang, L. (2000). Nonparametric lag selection for time series. *Journal of Time Series Analysis*, 21(4). 457-487.
- Uhlig, H. (1999). A Toolkit for Analysing Dynamic Stochastic Models easily. In Marimón, R. and Scott, A., editors, *Computational Methods for Study of Dynamic Economies*, chapter 3. Oxford University Press.
- Venners, B. (2002). A Conversation with Effective Java Author, Josh Bloch. Java-World.
- Vinod, D. H. (2000). Review of GAUSS for Windows, including its Numerical Accuracy. *Journal of Applied Econometrics*, 15(2):211–220.
- Peng Wu and Samuel P. Midkiff and Jose E. Moreira and Manish Gupta (1999). Efficient Support for Complex Numbers in Java. In *Java Grande*, pages 109–118.

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig ohne fremde Hilfe verfaßt und nur die angegebene Literatur und Hilfsmittel verwendet zu haben. Frühere Begutachtungen dieser Arbeit existieren nicht.

Ich bezeuge durch meine Unterschrift, dass meine Angaben über die bei der Abfassung meiner Dissertation benutzten Hilfsmittel, über die mir zuteil gewordene Hilfe sowie über frühere Begutachtungen meiner Dissertation in jeder Hinsicht der Wahrheit entsprechen.

Markus Krätzig

21.12.2004