Krätzig, Markus

Working Paper

A software framework for data based analysis

SFB 649 Discussion Paper, No. 2005-044

Provided in Cooperation with:
Collaborative Research Center 649: Economic Risk, Humboldt University Berlin

Suggested Citation: Krätzig, Markus (2005) : A software framework for data based analysis, SFB 649 Discussion Paper, No. 2005-044, Humboldt University of Berlin, Collaborative Research Center 649 - Economic Risk, Berlin

This Version is available at:
https://hdl.handle.net/10419/25063

SFB 649 Discussion Paper 2005-044

A Software Framework for Data Based Analysis

Markus Krätzig*

* Institute for Statistics and Econometrics,
Humboldt-Universität zu Berlin, Germany

This research was supported by the Deutsche Forschungsgemeinschaft through the SFB 649 "Economic Risk".

http://sfb649.wiwi.hu-berlin.de
ISSN 1860-5664

SFB 649, Humboldt-Universität zu Berlin
Spandauer Straße 1, D-10178 Berlin

A Software Framework for Data Based Analysis [1]

Markus Krätzig

Humboldt-Universität zu Berlin, Institute for Statistics and Econometrics

Current version: 29 August 2005

Abstract

This paper presents the software framework *JStatCom* which is geared towards the development of rich GUI clients for numerical procedures. [1] The concept is to solve all recurring tasks with the help of reusable Java components. Optionally, one can delegate the execution of special numerical algorithms to external programs, for example Gauss or Matlab. This way it is possible to reuse an already existing code base for numerical routines written in different programming languages and to link them with the Java world. A reference application for JStatCom is the econometric software package JMulTi, which will shortly be introduced. [2]

Key words:
Java, Object-Oriented Programming, Econometrics, Software Engineering

1 Introduction

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. This observation is equally valid in other fields were numerical algorithms are applied. However, in econometrics users also need a high level of user interaction with the software to insert a priori information to the statistical models under investigation.

[1] This research was supported by the Deutsche Forschungsgemeinschaft through the SFB 649 'Economic Risk'. Markus Krätzig: Humboldt-Universität zu Berlin, Department of Economics, Spandauer Str. 1, 10178 Berlin, Germany, email: mk@mk-home.de.

[1] GUI means Graphical User Interface.

[2] The URLs are *www.jstatcom.com* and *www.jmulti.com* respectively. Both projects are Open-Source.

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 the Oxmetrics family, 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, Matlab, SPSS, Stata, 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 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. However, some software products allow to design user defined modules, even with GUI support. One of the most advanced examples for this is the Oxmetrics family. Still, it will be shown that the presented approach with JStatCom can be considered as a generalization of that solution.

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.

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 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. This is where JStatCom steps in.

The paper is organized as follows. The next section describes the problem domain and mentions existing solutions. Section 3 gives an overview about the general structure of JStatCom. Section 4 introduces JMulTi, a reference application based on the framework. Section 5 concludes.

2 Problem Domain, Requirements, and Alternatives

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 can serve as a mediator between different procedure calls. Such a framework has to solve a number of problems that typically appear in this problem context. Therefore it seems worthwhile to try to develop reusable components that can help to make development more efficient. So far there are only isolated solutions for certain problems or languages, as for example described in Ashworth et al. (2003), but no attempt has been made to standardize the creation of GUIs for mathematical applications in a more general context. One exception is the web based approach MMM (Günther et al., 1997) which was developed as an architecture to share algorithms and computing resources via the Internet. User interaction was done via a browser interface. However, this solution was still not convenient enough for users and was therefore not widely used for empirical analysis.

The presented software framework JStatCom attempts 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 focused 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. 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 conceptual simplicity in the class design and by providing standardized ways to document and test applications based on it.

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 very 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. For example, object-oriented programming techniques are still not in widespread use for the development of econometric routines, although the additional effort to adopt these techniques would pay off quickly (Doornik, 2002). The reason might be that it requires a higher effort to lay out the structure of an object-oriented program, thus thinking more about the software itself instead of the problem.

However, the procedural, function-based programming style is often a sufficiently powerful way to solve computational problems occurring in econometrics. But it fails clearly when it comes to creating graphical user interfaces and when various different algorithms should be used together. The idea of JStatCom is to let scientists program in their preferred style, but to use object-oriented techniques to integrate existing algorithms. This way, domain-specific procedures can be reused and enhanced with a user interface.

The following subsections discuss the requirements aspects of the framework that have led to the major architectural decisions. One should note that most of the described requirements must be met for any GUI that should be used for interactive numerical calculations. Therefore the mentioned problems are solved in different ways in existing software packages. But JStatCom is the first framework that generalizes these solutions and makes them available as reusable classes. It should also be mentioned that JStatCom evolved from the experiences gained when developing the software JMulTi. Only in a second step, the framework was separated from the application. But the requirements were by then obvious from the development experiences and user requests.

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

2.2 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. [3] Components from other models, say from an STR model, do not have access to that repository, instead they use their own data pool. [4] 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

[3] Vector Autoregressive Models (Lütkepohl, 1991)
[4] Smooth Transition Regression (Teräsvirta, 1998)

debugging. Such a data control system must work automatically without any additional programming effort.

2.3 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 may 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. Furthermore, numerical calculations might take a long time. It is desirable that the GUI stays reactive even during intense computations, and that it possibly offers a way to cancel a running calculation. This makes thread-safety a requirement for all classes that might be used from the GUI thread and the thread in which the calculations run. Thus adequate synchronization is necessary.

2.4 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 enough, 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. [5]

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 is achieved by using Java as 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.

2.5 Key Design Features

2.5.1 Conceptional Simplicity

The framework should also be adopted by scientists, rather than pure 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 conceptional burden to understand and maintain them.

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

- representing scientific models with potentially many variables
- creating well structured, maintainable, feature-rich GUI components
- programming calls to external computational engines
- creating and integrating help documents for scientific models

[5] Users of the framework are developers who program a new GUI or application with the classes from JStatCom.

2.5.2 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.

The design of a framework for numerical calculations has to support the task of executing 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 a 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.

2.6 Existing Solutions for GUI Building

The idea to create user interfaces for scientific procedures is of course not new and there exist a number of approaches for that task. Most of them use special features of the respective language to set up predefined, customizable user interface components that are called from within the control flow of the program. This concept is used for example by Matlab and Xplore. Although it is very easy to create simple graphical applications with this strategy, it tends

to clutter GUI related code and algorithm code as the application is growing. Apart from that, the lack of data encapsulation increases interdependencies between different parts of the created software, such that it is getting harder to maintain and extend. There are many examples where Matlab has successfully been used to create stand-alone applications with a GUI, for example Uhlig (1999). But due to the growing complexity, those projects are limited in size and lifetime.

A different solution is provided by the Ox programming language with the extension `OxPack` (Doornik and Ooms, 2001). It is part of the Oxmetrics family of econometric software tools. 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 mentioned approach is that here an object-oriented design is provided to access 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 shows the relationship of the relevant classes for a hypothetical STR modelling class in a UML diagram. For clarity, the representation of those classes is simplified, not all public methods are shown.

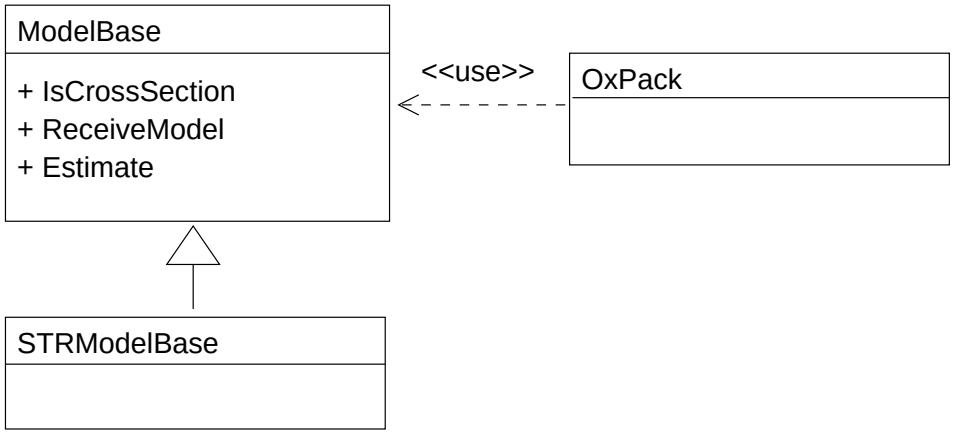


Fig. 1. Class diagram for an interactive Ox program

`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 method 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. Nevertheless, this restricts the applicability of the design to compatible modelling situations only. Models that require an extended set of features or that belong to a different problem domain would not fit into that framework. Apart from that, the behavior 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 behavior of the single steps. This means, that the flexibility of this approach to create interactive GUIs for various different models is somewhat limited.

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 another class, but that it keeps just a reference to instances of 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 can be derived from the `ModelBase` class, or that it might require special solutions that are not supported in a straightforward manner. 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. However, this lack of static structure is compensated by providing design guidelines that should help the developer to apply standardized solutions to heterogeneous models.

Compared to Ox with `OxPack`, JStatCom provides more flexibility to design applications based on it. It is not limited to a specific model setup anymore, not even a specific problem-domain, like econometrics. However, this comes of course at a price. Programming with JStatCom requires some knowledge in Java. Luckily, the Java programming language is increasingly popular and also more and more adopted by the science community, see for example Boisvert et al. (2001). There is an enormously rich documentation available and there is excellent tool support. In the following, selected implementation details of

JStatCom are presented and how the framework could be used. This text should motivate developers to give it a try. It might also help to decide when existing solutions are sufficiently powerful and when it will pay off to learn and use the presented approach.

3 JStatCom System Overview

This section aims at giving a quick overview of the main features and the basic workings of the framework. It is by no means a complete documentation or specification. For a deeper understanding, the API documentation in javadoc format as well as the architecture documentation is required (Krätzig, 2004).

JStatCom is a software 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). This 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. Some of the solutions presented in this section have already been sketched in Benkwitz (2002), where the first prototype of the system was described.

Figure 2 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 setup a test for the software. The GUI developer can focus on the Java side, taking the algorithms as given. JStatCom serves as an architectural layer that handles all tasks that are common to applications in the given problem domain, which is econometrics for the current example.

The collaboration of components that make up an arbitrary runnable application is shown in Figure 3. The application, for example JMulTi, uses the framework, which itself manages the communication to an external execution engine. [6] Algorithm implementations have to be provided as resources for the respective computational engine.

The top level elements of the system are given in Table 1. Each element corresponds to a subsystem with coherent functionality that can be separately looked at. Developers use the components to lay out the user interface, the

[6] JMulTi is the reference application for JStatCom. The URL is *www.jmulti.de*.

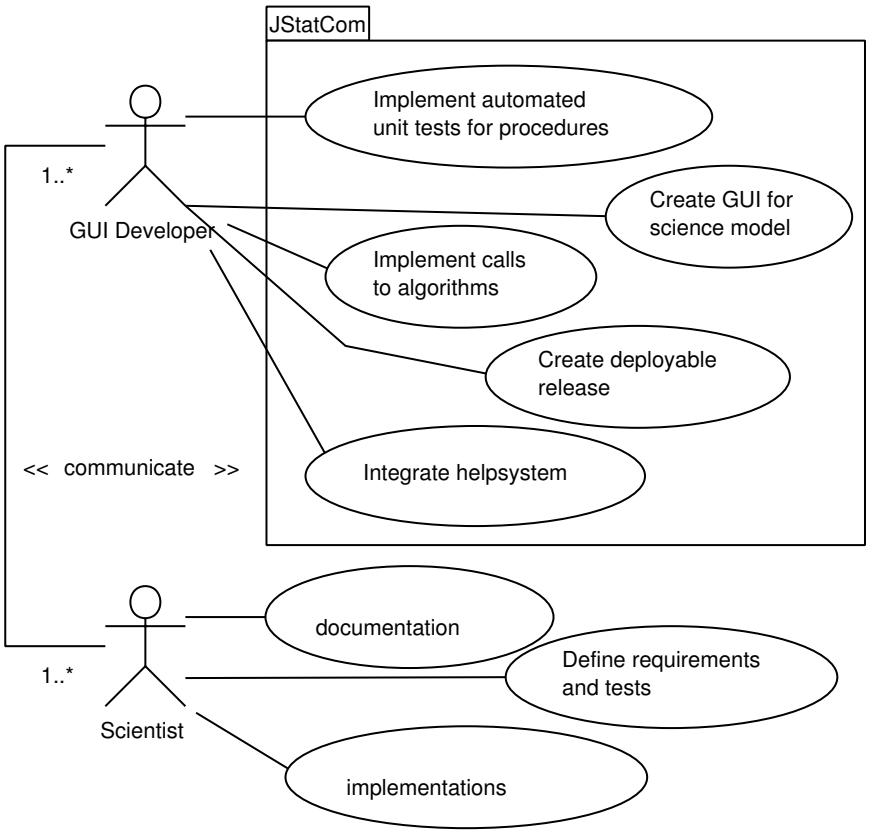


Fig. 2. Use cases for JStatCom

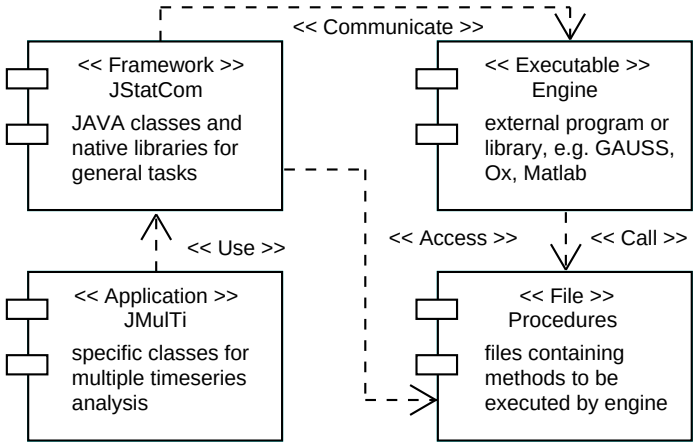


Fig. 3. Components of JStatCom

data model to represent variables of the model, and the Engine System to communicate to the respective engine to invoke algorithms. All elements can be further decomposed into classes or other subsystems. However, for the sake of clarity, only the Data Model and the Engine system are described in greater detail.

Element Name	Element Responsibility
Data Model	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	Contains classes to support file handling and the **Data Import System**. It also provides a logging facility.
Project	Manages storing and retrieving data and the state of analysis modules to and from XML project files.
Time Series	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	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.

Table 1: Elements of JStatCom

3.1 Data Model

JStatCom needs to represent data internally, because it maintains inputs and results of numerical 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. This mechanism is hidden from the developer and managed automatically by the engine implementations.

3.1.1 Tpye System

The framework uses a metadata model to achieve the desired flexibility. Core attributes are standardized for all data types by defining a very general interface `JSCData`, which all specific types must implement. This interface does only specify 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. An alternative 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.

Figure 4 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. It should be noted that the implemented types are 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 computational engine, which is especially suited and optimized for that purpose.

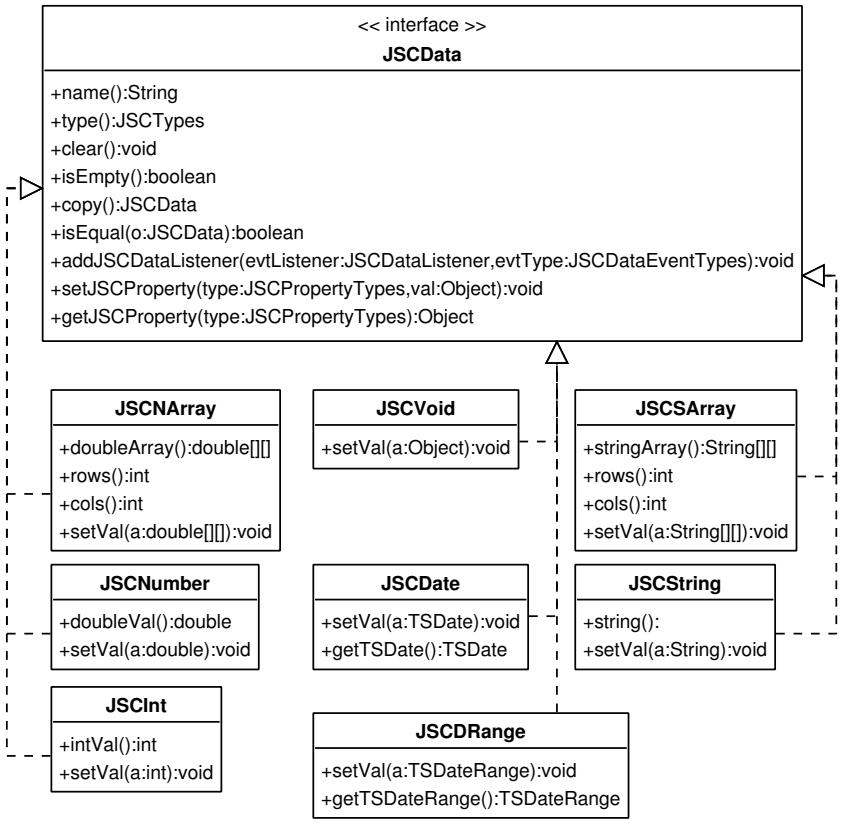


Fig. 4. Type System

The following small code example demonstrates how instances of different types can be created in Java. A special feature is that every 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. The code also shows, how different types can be treated uniformly as a `JSCData` array. This can greatly simplify method signatures. However, if the type-specific functionality is needed, then a cast to the respective implementation class is necessary. A save way to do this is to check the type before.

```
// 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 index = new JSCInt("i", 3);

// all data can be treated uniformly as JSCData
JSCData[] args = new JSCData[]{y, start, index};
```

```
// if the concrete implementation is needed, casting is necessary
// the type can be checked before
JSCTypes type = args[0].type();
if (type == JSCTypes.NARRAY){
    JSCNArray yRef = (JSCNArray) args[0];
    System.out.println(yRef.doubleAt(0,0));
}
```

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-save, it should inform listeners about changes in the data, it should be XML serializable and it should 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.

3.1.2 Symbol Management

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 analyzed, 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 number of shared variables can become quite large. In JMulTi there are 37 shared symbols for representing all information for a VAR system. For more complex models this number increases.

The user interface typically consists of 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.

Gamma et al. (1995) suggest the *State* pattern in this case. A State could be implemented as a class that represents a model, say `VARState`. This state object could then be shared among all participating components. On a first glance this would be the solution that resembles best the spirit of object-oriented programming. However, the drawback of this approach is that the developer would need to create a `VARState` state class first and she would then

have to find a mechanism to publish it to all components that need access to it. The hypothetical `VARState` class would become quite large soon, because it would have to store also the names of variables, the estimation method and various other settings. Apart from the effort of creating and maintaining such a class, this procedure does not generate a standard way of creating GUIs for an arbitrary model, because it would most likely lead to different solutions for each model that is implemented. The quality, extendability and maintainability of model implementations would differ largely. Therefore it would be desirable to have a straightforward way to represent and share the state of just any possible model without the need to think about how to create state classes and how to share them. This would also be a good example not only of class reuse, but of design reuse, which is one of the major benefits of programming with a framework.

Figure 5 gives a simplified overview of the Symbol Management system which is the JStatCom solution to address the raised issues. It consists of a 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 the 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 variables. The referenced values, instances of `JSCData`, can be changed efficiently during runtime, but not the type. 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`. The `SymbolTable` can represent the state of arbitrary models as an aggregation of symbols of different types. It is therefore much more general, but also less specific than the previously mentioned `VARState` class. All shared global data should reside in a symbol table, which is then accessed by the components of a model.

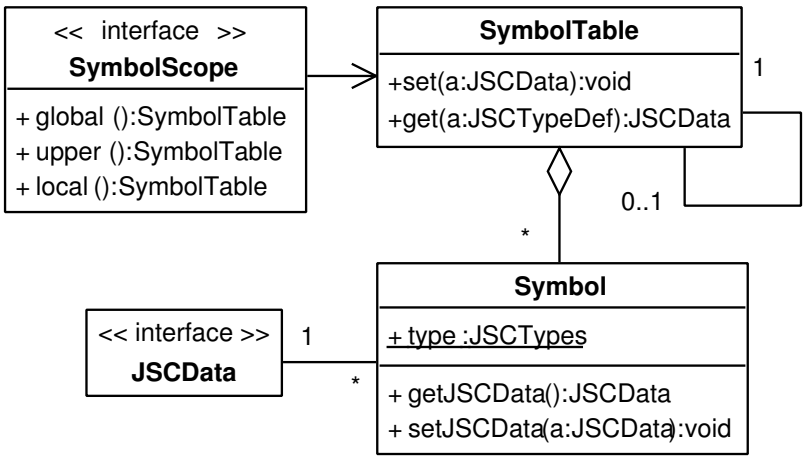


Fig. 5. Accessing shared data repositories

One might ask, whether this is not just another way of introducing global data. In a way it is, but there is another part of the Symbol Management system which allows for fine-grained definition of access scopes. The question is, which components can use a certain symbol table? 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.

To be more specific, Figure 6 shows, how the `SymbolScope` interface is implemented by components of the model. Every model should be implemented with a `ModelFrame` as the top level component. This can be the starting point for any application based on JStatCom. 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 component hierarchy for this. Symbol tables are assigned as follows:

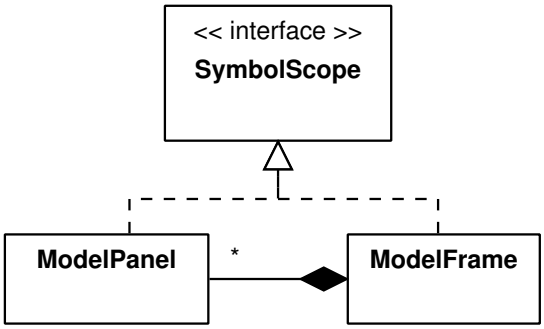


Fig. 6. SymbolScope inheritance

- `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

This process is done automatically and developers only need to understand that `ModelPanel` instances can be used to define access scopes. One could also think of other possible implementations of `SymbolScope`, reflecting different hierarchical schemes. However, for the purpose of GUI building this solution has proven to be very fruitful.

One might be tempted to compare `ModelFrame` to the `ModelBase` class in Ox that was introduced in Section 2.6. The only similarity is that both classes should be subclassed to create a new model. `ModelFrame` does not provide any model specific functionality, except the access methods to the symbol table. No specific structure for components, behaviour, or model types is imposed. But, theoretically, one could implement the functionality of `ModelBase` in a specific `ModelFrame` implementation to provide further standardization for a distinct problem domain. Therefore JStatCom can be considered as a more general approach which could nest more specialized solutions similar to the one provided by Ox.

Figure 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`, and `global` from `SymbolScope`.

A snapshot of the object structure at runtime is presented in Figure 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 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

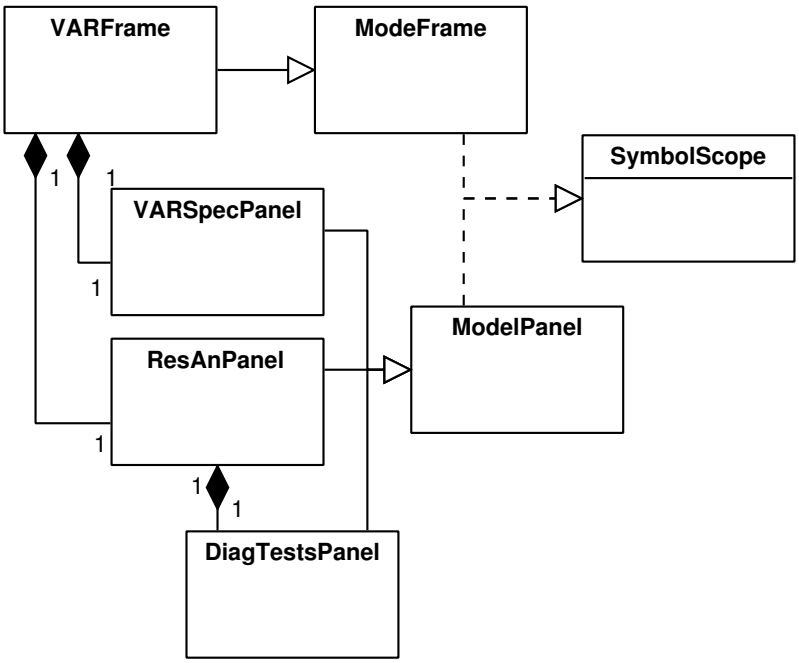


Fig. 7. Class structure of a hypothetical VAR frame

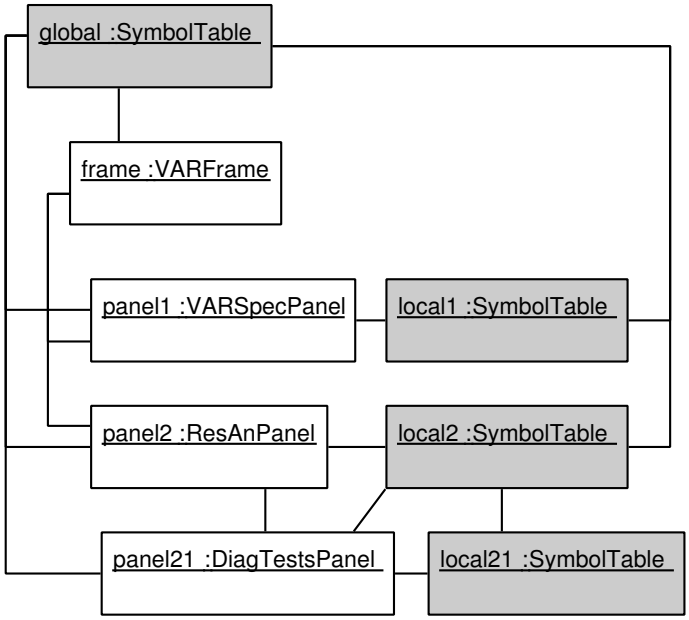


Fig. 8. Snapshot of model objects and shared data with different scopes

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`.

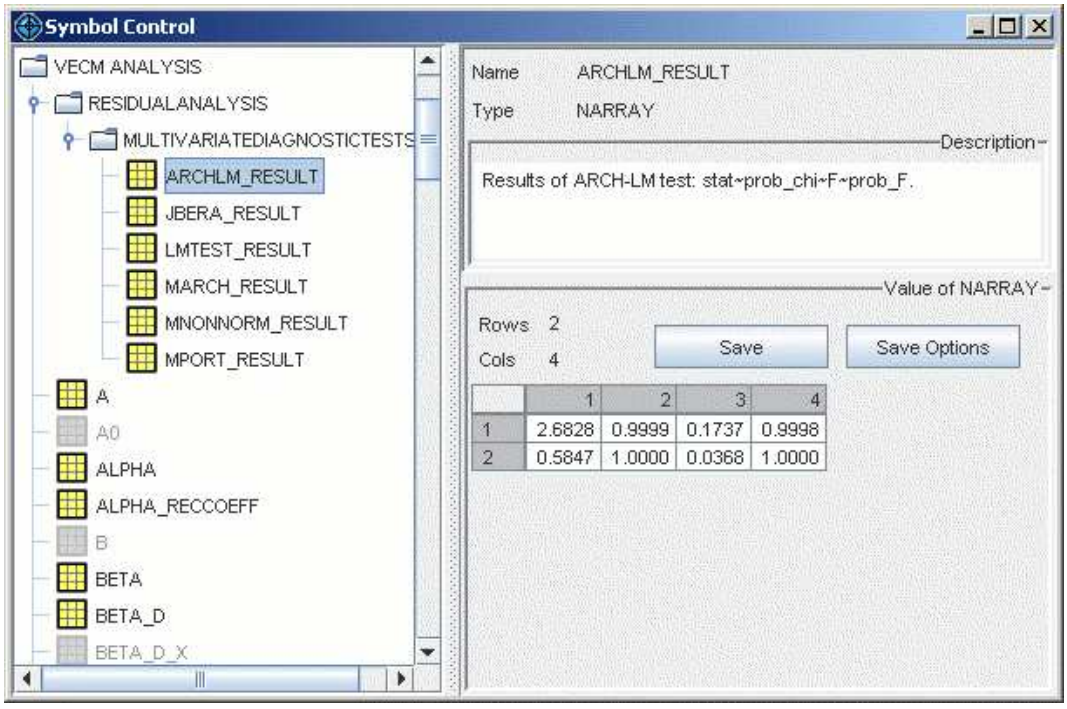


Fig. 9. Screenshot of symbol frame with selected `NARRAY`

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, which is another subsystem of JStatCom that provides access to variables that are currently used. Figure 9 presents a screenshot of the graphical component. A description is omitted here, but 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.

The following small Java code example should demonstrate the workings of the Symbol Management system. It corresponds to the class diagram in Figure 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. JStatCom provides several special components for that purpose, but they are not described here. 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 resAnPanel;
    private VARSpecPanel vARSpecPanel;
    ...
    // constructor
    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
  private void setEstimationSettings(){
    global().set(new JSCString("EstimationMethod", "OLS"));
    ... // more variables to set, omitted for clarity
  }
 } // end VARSpecPanel

// panel for residual analysis
public class ResAnPanel extends ModelPanel {
    public DiagTestsPanel diagTestsPanel;
    ...
    // constructor
    public ResAnPanel(){
      super();
      setResidNames();
      // 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 JSCSArray("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().get("EstimationMethod")
                                .getJSCString();
    JSCSArray resNames = upper().get("ResNames").getJSCSArray();
    ... // more variables to be retrieved, omitted for clarity
    ... // 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 that is also displayed in the Symbol Control. Using this way of defining shared data helps greatly to manage even large GUI systems with many variables.

3.2 Engine System

This section introduces the system for communicating to different execution engines. Typically these engines rely on external resources, which means that extra software packages or libraries must be installed. For example, for the Ox engine the installation of Ox console is required together with the extra packages that are used. There are other software vendors who provide redistributable stand-alone versions of their computational engine, for example Aptech with the Gauss Runtime Engine. The advantage is that users do not need any extra packages to be installed on their computer. The software JMulTi uses that engine for all computing tasks.

The framework tries to provide access to different engine implementations via a unified interface. Figure 10 presents the complete interface `Engine` and all implementations currently available. Clients should use the engine only via its abstract implementation, thus making similar calls for every implementation. According to Gamma et al. the used pattern is a *Facade*. 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,

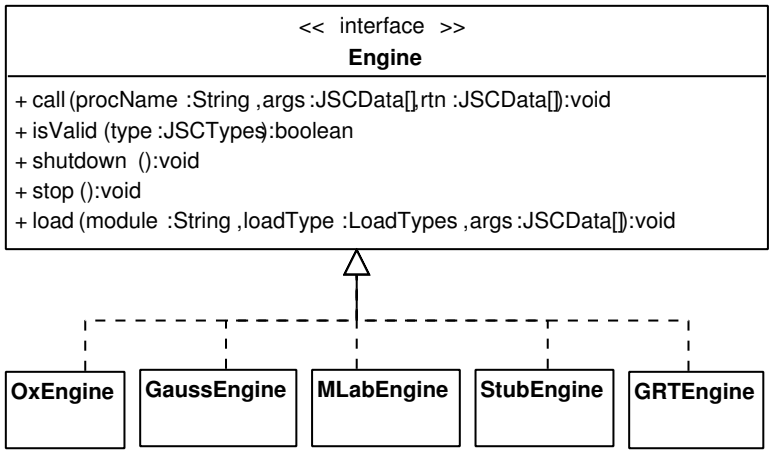


Fig. 10. Engine interface and available implementations

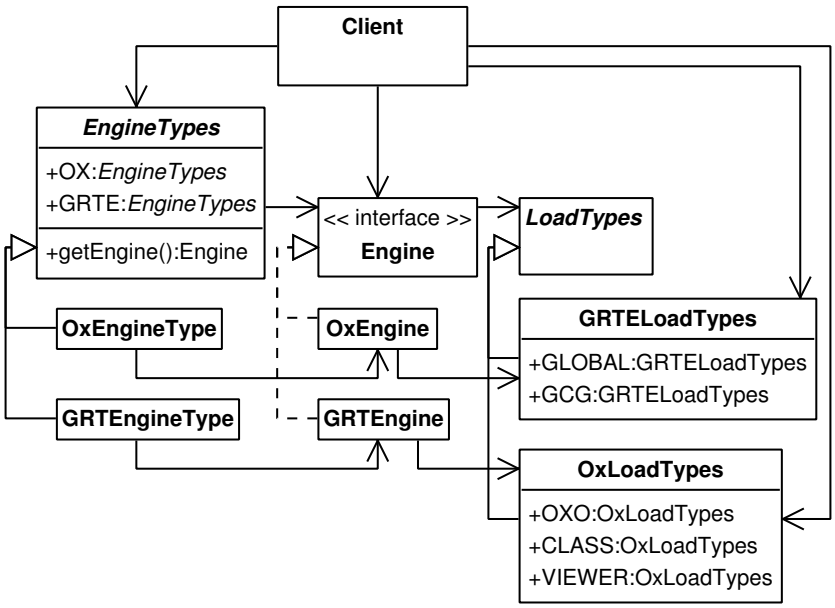


Fig. 11. Client using the Engine system

the engine interface provides the parametrized function `load` to address these issues. The method takes a parameter of type `LoadTypes` that defines the specific action to carry out.

Figure 11 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 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. There exist Java interfaces for both tools already, the task of creating engine implementations is therefore merely to manage the type conversion between JStatCom data types and engine specific types, as well as to handle configuration settings. This undertaking is supported by the fact that most tool vendors supply programming interfaces to control the respective software from an external application, examples are the Ox C-API, the Gauss Runtime Engine, or the J/Link package for Mathematica, to name just a few.

3.2.1 Introductory 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`. Resources contain the algorithm implementations for an engine, and there are special directories where JStatCom looks for them. By convention, this is a subdirectory which starts with a `j` followed by the name of the engine, thus `jox`, `jgrte`, `jgauss`, `jstub`, and `jmlab`. 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`. The Engine system has an elaborate configuration management, which is used to gather environment settings from a configuration file, and, if something is missing or wrong, from the user directly. The required settings vary from engine to engine. But all engines store information in a file `engine_config.xml` in the respective resource directory.

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:

```
// EngineTypes stores all available engine types
// ox is an instance of OxEngine, but the client
// does not use this information
Engine ox = EngineTypes.OX.getEngine();

// parametrized call to load with OxLoadTypes referenced,
// puts mymodule.ox(o) in Ox workspace, no arguments
ox.load("mymodule", OxLoadTypes.OXO, null);

// another load call, equivalent to decl x = new MyClass(3);
// MyClass must be defined in mymodule.ox(o)
// x is the object from which member functions can be called
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
```

This code snippet has not created any user interface components, but demonstrates how the Type System together with the Engine system could be used to make a call to Ox. The Symbol Management is not involved here, because no data is shared. It is straightforward to replace the example Ox class with a real world counterpart that does some computation not easily done in Java. Similar examples for different engines can be found in Krätzig (2004).

4 A short Introduction to JMulTi

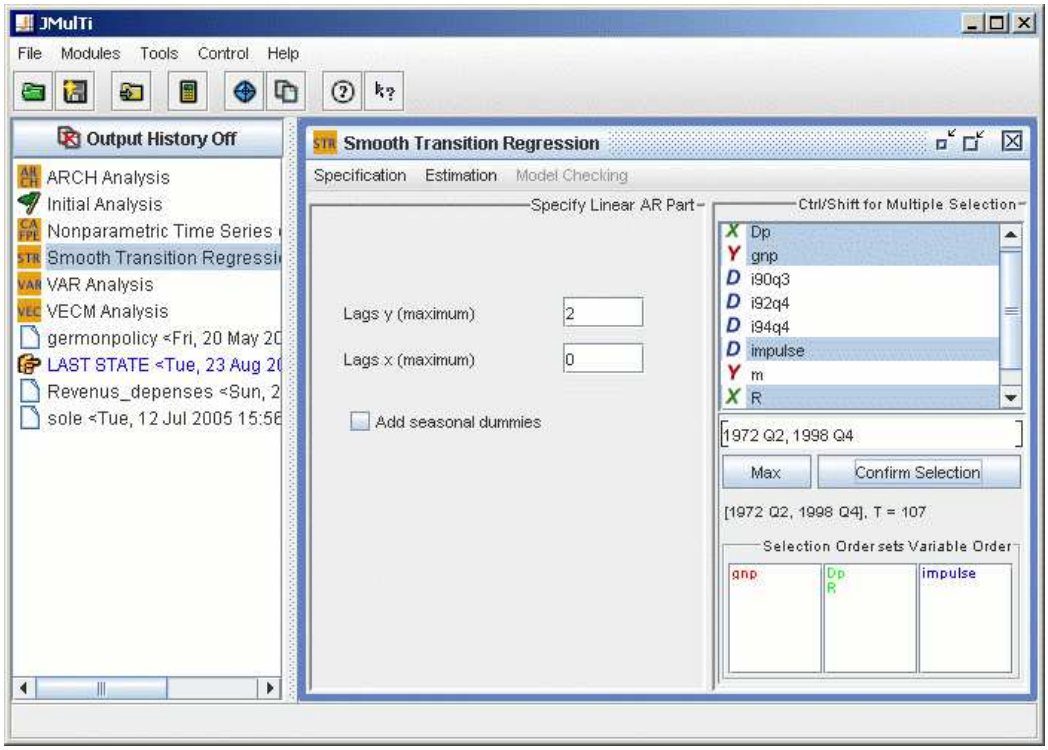


Fig. 12. Running instance of JMulTi

The framework JStatCom 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.

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. In May 2004 the project went Open-Source and user feedback has significantly increased since then. It is also hoped that interested developers for econometric routines will join.

4.1 General Setup

Figure 12 shows the main window of JMulTi with the project explorer and the various analysis modules. The general usage of program is pretty much predetermined by the layout of the framework JStatCom. 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 data import, project management, a time series calculator, the symbol control system, a logging facility, and access to the help system.
- *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.

JMulTi uses the GRTE or the Gauss engine. The Gauss engine is only used for development and debugging purposes. It is possible to switch between those two engines via a command line option.

Table 2 presents all modules that have been implemented so far. 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.

Analysis Module	Description
Initial	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	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	Specification and estimation of VEC models with impulse response analysis, diagnostic checks, forecasting, and more. Also offers the possibility to estimate SVEC models.
ARCH	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	Specification and estimation of STR models, as well as nonlinearity tests. All parts of the estimated STR model can be plotted.
Nonparametric	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 2: Modules of JMulTi

It would always be possible to add newly defined modules without interfering with the existing ones. In the remaining parts of this section, only the Initial Analysis and the VAR module are described in some detail. It is hoped that also readers who are not in any way familiar with econometrics understand broadly how the software works and that the concept might well be translated to other problem domains.

4.2 Initial Analysis

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 13 shows the textual output of the computation of the AC and PAC functions for a selected series.

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 14. 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 15 has a screenshot of one of the implemented cointegration tests.

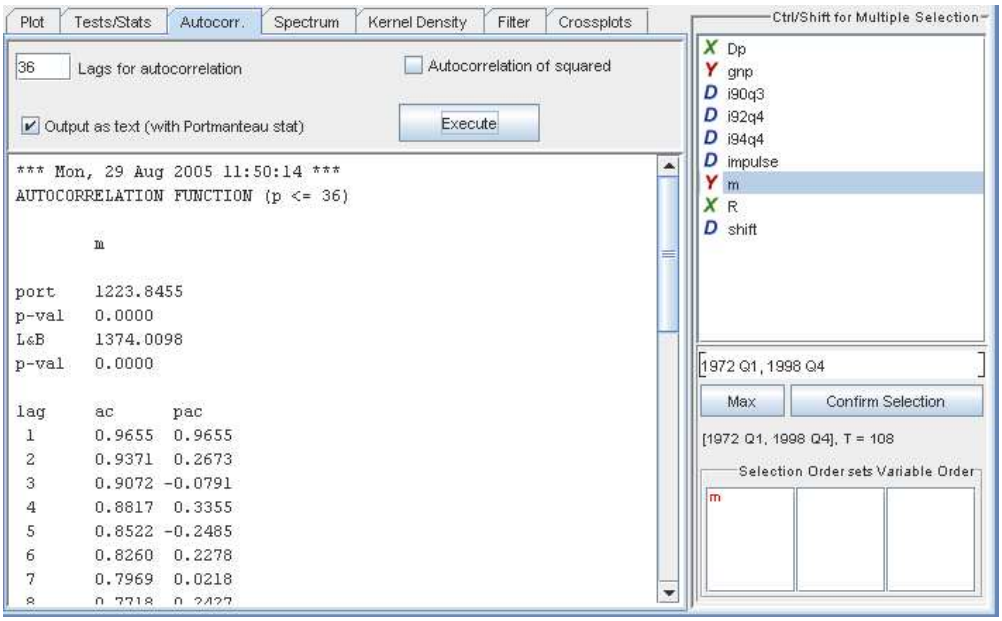


Fig. 13. Screenshot of workbench with autocorrelation panel

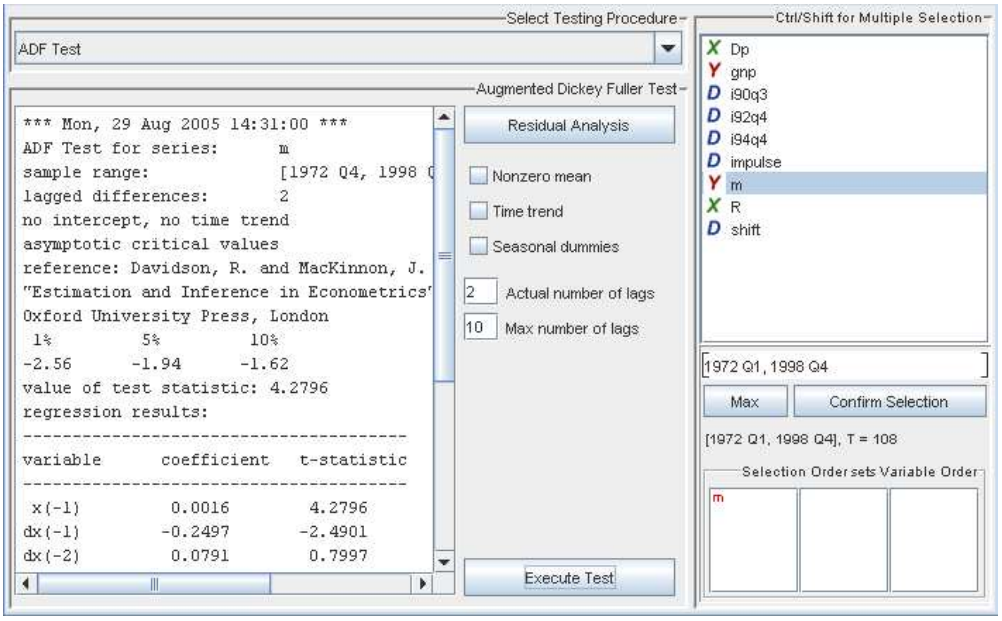


Fig. 14. Screenshot of ADF unit root test panel

4.3 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

$$y_t = A_1 y_{t-1} + \cdots + A_p y_{t-p} + B_0 x_t + \cdots + B_q x_{t-q} + CD_t + u_t, \qquad (1)$$

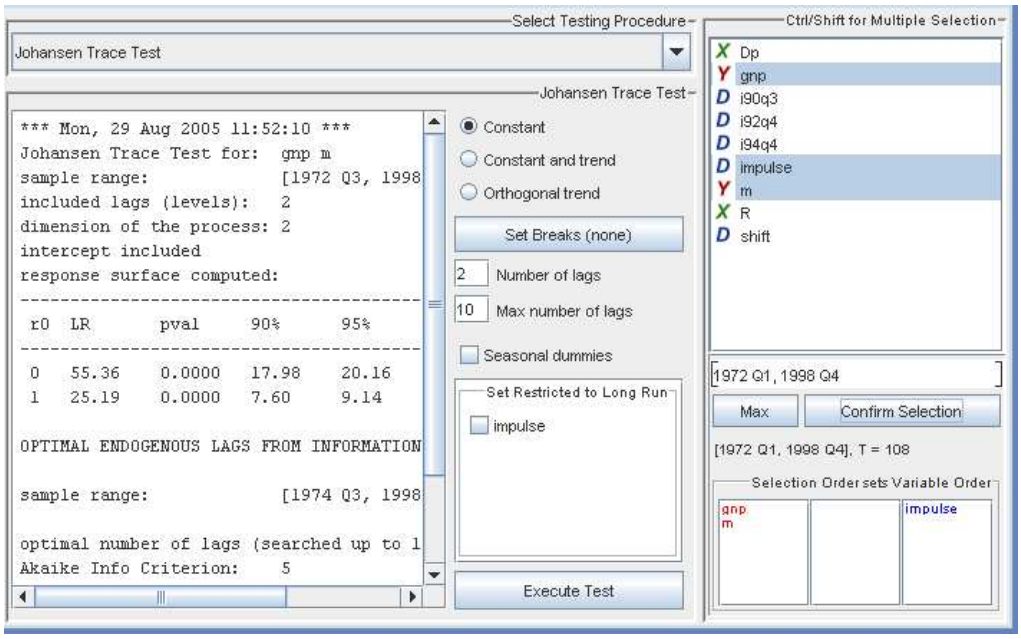


Fig. 15. Screenshot of Johansen cointegration test panel

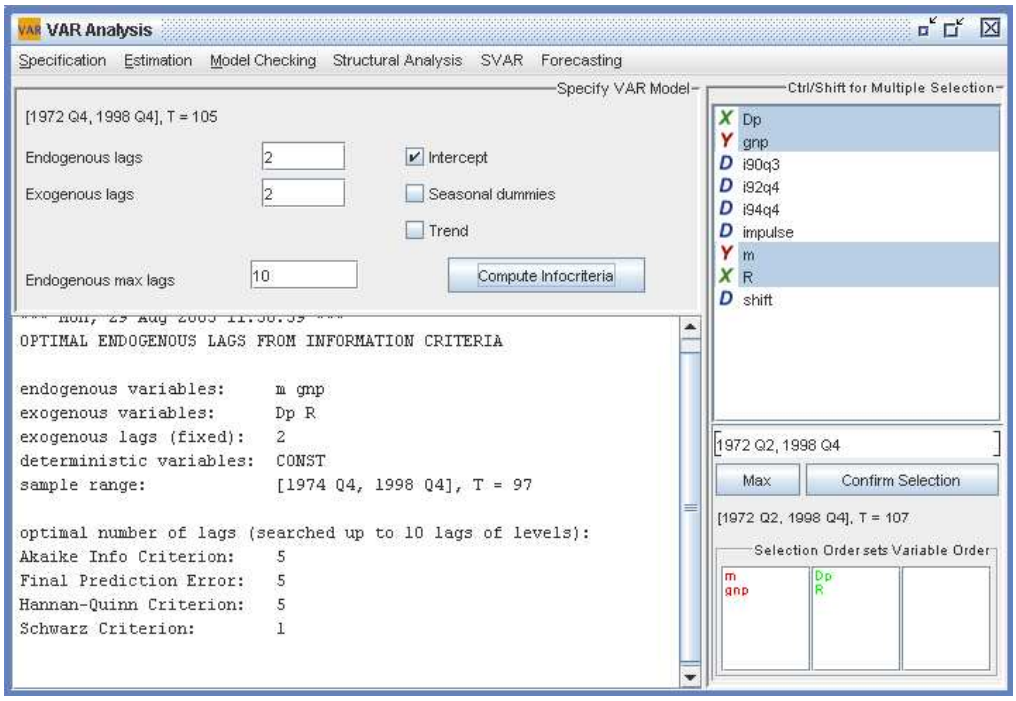


Fig. 16. Screenshot of specification panel for the VAR analysis

where $y_t = (y_{1t}, \ldots, y_{Kt})'$ is a vector of K observable endogenous variables, $x_t = (x_{1t}, \ldots, x_{Mt})'$ 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 16.

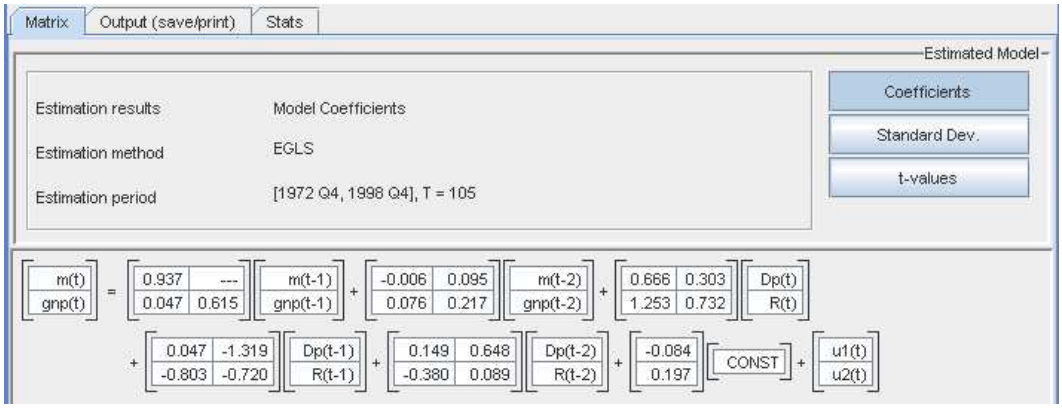


Fig. 17. Screenshot of estimation panel for the VAR analysis

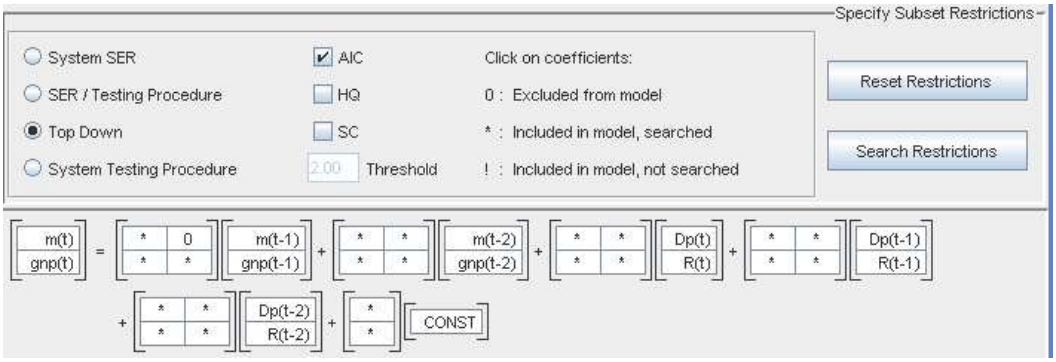


Fig. 18. Screenshot of manual/automatic subset specification for the VAR analysis

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 (1), see Figure 17.

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 16). 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 18. It is possible to set subset restrictions 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 (Brüggemann and Lütkepohl (2001)).

It should be noted that the general VAR model in Equation (1) 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 16. 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 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 18 presents the panel for applying model reduction strategies.

Once the model is estimated, see Figure 17, 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. Finally, forecasting of the levels and the undifferenced series is provided as an option to the user.

In a similar way also the remaining modules of JMulTi are implemented. Because they represent different theoretical models, the underlying procedures and GUI components differ. However, the general structure is always very similar. There are several GUI components that access a common shared data repository with the data for the model and with results from previous analysis steps. Algorithms are executed via the computational engine that is employed.

5 Conclusion

It has been argued that JStatCom provides standard solutions to recurring tasks. This can be a big advantage over other approaches for GUI creation, because it allows to implement a wide range of models with similar programming techniques and with a similar design. 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, it might be necessary to develop new types to represent data specific to some field in engineering or physics.

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.

References

Ashworth, M., Allan, R., Müller, C., van Dam, H., Smith, W., Hanlon, D., Searly, B. and Sunderland, A. (2003). Graphical user environments for scientific computing, *Technical report*, Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Warrington.
URL: *http://www.ukhec.ac.uk/publications/reports/guienv.pdf*

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*, 1st edn, Addison-Wesley.

Benkwitz, A. (2002). *The Software JMulTi: Concept, Development and Application in VAR Analysis*, Dissertation, Humboldt-Universität zu Berlin.

Bloch, J. (2001). *Effective Java*, Addison-Wesley.

Boisvert, R. F., Moreira, J., Philippsen, M. and Pozo, R. (2001). Numerical Computing in Java, *Computing in Science and Engineering* **3**(2): 18–24.
URL: *http://citeseer.ist.psu.edu/409642.html*

Boisvert, R. F. and Tang, P. T. P. (eds) (2001). *The Architecture of Scientific Software, IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software, October 2-4, 2000, Ottawa, Canada*, Vol. 188 of *IFIP Conference Proceedings*, Kluwer.

Brüggemann, R. and Lütkepohl, H. (2001). Lag selection in subset VAR models with an application to a U.S. monetary system, *in* R. Friedmann, L. Knüppel and H. Lütkepohl (eds), *Econometric Studies: A Festschrift in Honour of Joachim Frohn*, LIT Verlag, Münster, pp. 107–128.

Deutsch, L. P. (1989). Design reuse and frameworks in the Smalltalk-80 system, *in* T. J. Biggerstaff and A. J. Perlis (eds), *Software Reusability, Volume II: Applications and Experience*, Addison-Wesley, Reading, MA, pp. 57–71.

Dijkstra, E. W. (1969). Structured programming. circulated privately.
URL: *http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF*

Doornik, J. (2002). Object-oriented Programming in Econometrics and Statistics using Ox: A Comparison with C++, Java and C#, *in* S. Nielsen (ed.), *Programming Languages and Systems in Computational Economics and Finance*, Dordrecht: Kluwer Academic Publishers, pp. 115–147.

Doornik, J. and Ooms, M. (2001). *Introduction to Ox*, Timberlake Consultants Press, London.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Günther, O., Müller, R., Schmidt, P., Bhargava, H. and Krishnan, R. (1997). MMM: A WWW-based approach for sharing statistical software modules,

IEEE Internet Computing **1**(3): 59–68.

Johnson, R. E. and Foote, B. (1988). Designing reusable classes, *Journal of Object-Oriented Programming* **1**(2): 22–35.

Krätzig, M. (2004). *A Software Framework for Data Based Analysis*, Dissertation, Humboldt-Universität zu Berlin.
URL: *http://edoc.hu-berlin.de/dissertationen/kraetzig-markus-2005-02-04/PDF/Kraetzig.pdf*

Lütkepohl, H. (1991). *Introduction to multiple time series analysis*, Springer Verlag, Berlin.

Lütkepohl, H. and Krätzig, M. (eds) (2004). *Applied Time Series Econometrics*, Cambridge University Press, Cambridge.

Teräsvirta, T. (1998). Modeling economic relationships with smooth transtition regressions, *in* A. Ullah and D. Giles (eds), *Handbook of Applied Economic Statistics*, Dekker, New York, pp. 229–246.

Uhlig, H. (1999). A Toolkit for Analysing Dynamic Stochastic Models easily, *in* R. Marimom and A. Scott (eds), *Computational Methods for Study of Dynamic Economies*, Oxford University Press, chapter 3.

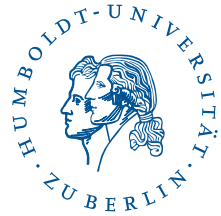
SFB 649 Discussion Paper Series

For a complete list of Discussion Papers published by the SFB 649, please visit http://sfb649.wiwi.hu-berlin.de.

001 "Nonparametric Risk Management with Generalized Hyperbolic Distributions" by Ying Chen, Wolfgang Härdle and Seok-Oh Jeong, January 2005.
002 "Selecting Comparables for the Valuation of the European Firms" by Ingolf Dittmann and Christian Weiner, February 2005.
003 "Competitive Risk Sharing Contracts with One-sided Commitment" by Dirk Krueger and Harald Uhlig, February 2005.
004 "Value-at-Risk Calculations with Time Varying Copulae" by Enzo Giacomini and Wolfgang Härdle, February 2005.
005 "An Optimal Stopping Problem in a Diffusion-type Model with Delay" by Pavel V. Gapeev and Markus Reiß, February 2005.
006 "Conditional and Dynamic Convex Risk Measures" by Kai Detlefsen and Giacomo Scandolo, February 2005.
007 "Implied Trinomial Trees" by Pavel Čížek and Karel Komorád, February 2005.
008 "Stable Distributions" by Szymon Borak, Wolfgang Härdle and Rafal Weron, February 2005.
009 "Predicting Bankruptcy with Support Vector Machines" by Wolfgang Härdle, Rouslan A. Moro and Dorothea Schäfer, February 2005.
010 "Working with the XQC" by Wolfgang Härdle and Heiko Lehmann, February 2005.
011 "FFT Based Option Pricing" by Szymon Borak, Kai Detlefsen and Wolfgang Härdle, February 2005.
012 "Common Functional Implied Volatility Analysis" by Michal Benko and Wolfgang Härdle, February 2005.
013 "Nonparametric Productivity Analysis" by Wolfgang Härdle and Seok-Oh Jeong, March 2005.
014 "Are Eastern European Countries Catching Up? Time Series Evidence for Czech Republic, Hungary, and Poland" by Ralf Brüggemann and Carsten Trenkler, March 2005.
015 "Robust Estimation of Dimension Reduction Space" by Pavel Čížek and Wolfgang Härdle, March 2005.
016 "Common Functional Component Modelling" by Alois Kneip and Michal Benko, March 2005.
017 "A Two State Model for Noise-induced Resonance in Bistable Systems with Delay" by Markus Fischer and Peter Imkeller, March 2005.

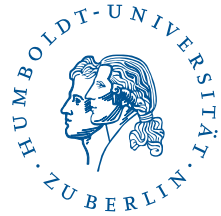
SFB 649, Spandauer Straße 1, D-10178 Berlin
http://sfb649.wiwi.hu-berlin.de

This research was supported by the Deutsche Forschungsgemeinschaft through the SFB 649 "Economic Risk".



018 "Yxilon – a Modular Open-source Statistical Programming Language" by Sigbert Klinke, Uwe Ziegenhagen and Yuval Guri, March 2005.
019 "Arbitrage-free Smoothing of the Implied Volatility Surface" by Matthias R. Fengler, March 2005.
020 "A Dynamic Semiparametric Factor Model for Implied Volatility String Dynamics" by Matthias R. Fengler, Wolfgang Härdle and Enno Mammen, March 2005.
021 "Dynamics of State Price Densities" by Wolfgang Härdle and Zdeněk Hlávka, March 2005.
022 "DSFM fitting of Implied Volatility Surfaces" by Szymon Borak, Matthias R. Fengler and Wolfgang Härdle, March 2005.
023 "Towards a Monthly Business Cycle Chronology for the Euro Area" by Emanuel Mönch and Harald Uhlig, April 2005.
024 "Modeling the FIBOR/EURIBOR Swap Term Structure: An Empirical Approach" by Oliver Blaskowitz, Helmut Herwartz and Gonzalo de Cadenas Santiago, April 2005.
025 "Duality Theory for Optimal Investments under Model Uncertainty" by Alexander Schied and Ching-Tang Wu, April 2005.
026 "Projection Pursuit For Exploratory Supervised Classification" by Eun-Kyung Lee, Dianne Cook, Sigbert Klinke and Thomas Lumley, May 2005.
027 "Money Demand and Macroeconomic Stability Revisited" by Andreas Schabert and Christian Stoltenberg, May 2005.
028 "A Market Basket Analysis Conducted with a Multivariate Logit Model" by Yasemin Boztuğ and Lutz Hildebrandt, May 2005.
029 "Utility Duality under Additional Information: Conditional Measures versus Filtration Enlargements" by Stefan Ankirchner, May 2005.
030 "The Shannon Information of Filtrations and the Additional Logarithmic Utility of Insiders" by Stefan Ankirchner, Steffen Dereich and Peter Imkeller, May 2005.
031 "Does Temporary Agency Work Provide a Stepping Stone to Regular Employment?" by Michael Kvasnicka, May 2005.
032 "Working Time as an Investment? – The Effects of Unpaid Overtime on Wages, Promotions and Layoffs" by Silke Anger, June 2005.
033 "Notes on an Endogenous Growth Model with two Capital Stocks II: The Stochastic Case" by Dirk Bethmann, June 2005.
034 "Skill Mismatch in Equilibrium Unemployment" by Ronald Bachmann, June 2005.

This research was supported by the Deutsche Forschungsgemeinschaft through the SFB 649 "Economic Risk".



035 "Uncovered Interest Rate Parity and the Expectations Hypothesis of the Term Structure: Empirical Results for the U.S. and Europe" by Ralf Brüggemann and Helmut Lütkepohl, April 2005.
036 "Getting Used to Risks: Reference Dependence and Risk Inclusion" by Astrid Matthey, May 2005.
037 "New Evidence on the Puzzles. Results from Agnostic Identification on Monetary Policy and Exchange Rates." by Almuth Scholl and Harald Uhlig, July 2005.
038 "Discretisation of Stochastic Control Problems for Continuous Time Dynamics with Delay" by Markus Fischer and Markus Reiss, August 2005.
039 "What are the Effects of Fiscal Policy Shocks?" by Andrew Mountford and Harald Uhlig, July 2005.
040 "Optimal Sticky Prices under Rational Inattention" by Bartosz Maćkowiak and Mirko Wiederholt, July 2005.
041 "Fixed-Prize Tournaments versus First-Price Auctions in Innovation Contests" by Anja Schöttner, August 2005.
042 "Bank finance versus bond finance: what explains the differences between US and Europe?" by Fiorella De Fiore and Harald Uhlig, August 2005.
043 "On Local Times of Ranked Continuous Semimartingales; Application to Portfolio Generating Functions" by Raouf Ghomrasni, June 2005.
044 "A Software Framework for Data Based Analysis" by Markus Krätzig, August 2005.

This research was supported by the Deutsche Forschungsgemeinschaft through the SFB 649 "Economic Risk".