Creating user interfaces for econometric routines with JStatCom: An example for Ox

Markus Krätzig

Humboldt-Universität zu Berlin, Institute for Statistics and Econometrics

mk@mk-home.de

First version: 16 June 2004
Current version: 31 July 2004

Abstract

JStatCom is a new software framework that simplifies the creation of graphical user interface components for mathematical procedures. [1] It is written in Java and offers a coherent approach to creating applications for data based analysis. Programming with JStatCom is efficient, because existing algorithms written in various popular matrix languages can be reused with little or no changes required. The paper shows how this system can be applied to provide an existing Ox program with a feature-rich graphical user interface with relatively little effort. One of the advantages of the presented approach is the strict separation of user interface and algorithm code. Although the framework provides standard solutions for common tasks, it can be extended and customized in various directions.

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

1 Introduction

The aim of this work is to present a new approach to creating software for analyzing data with statistical methods. It mainly addresses developers who intend to program graphical user interfaces for mathematical algorithms. 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

[1] The URL is *www.jstatcom.com*.

to create scalable applications that do not tend to become more complicated and error-prone as more features are added, and thus avoiding the common entropy problems (Bianchi et al., 2001). This is achieved by exploiting current developments in software engineering, like Design Patterns (Gamma et al., 1995) and unit testing (Beck, 1999).

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 GUIs for mathematical applications in a more general context.

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 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. 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. It only 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.

2 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). 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. The UML notation is a widely accepted standard to describe software systems, see Booch et al. (1999) for an exhaustive discussion.

Subclassing means, that all functionality from a superclass is inherited, but that behavior 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

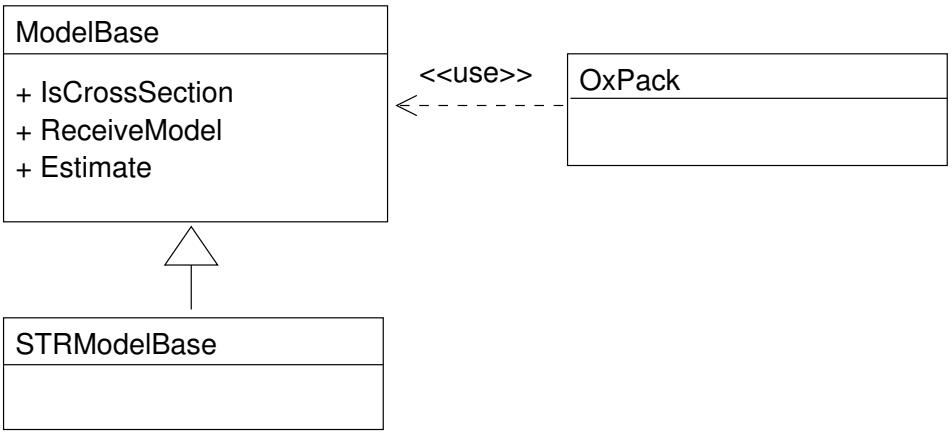

Fig. 1. Class diagram for an interactive Ox program

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. The following sections should give an introduction to the workings of that framework and should motivate developers to give it a try. This text 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. The first is part of the JStatCom distribution, the second is still under development at this moment.

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

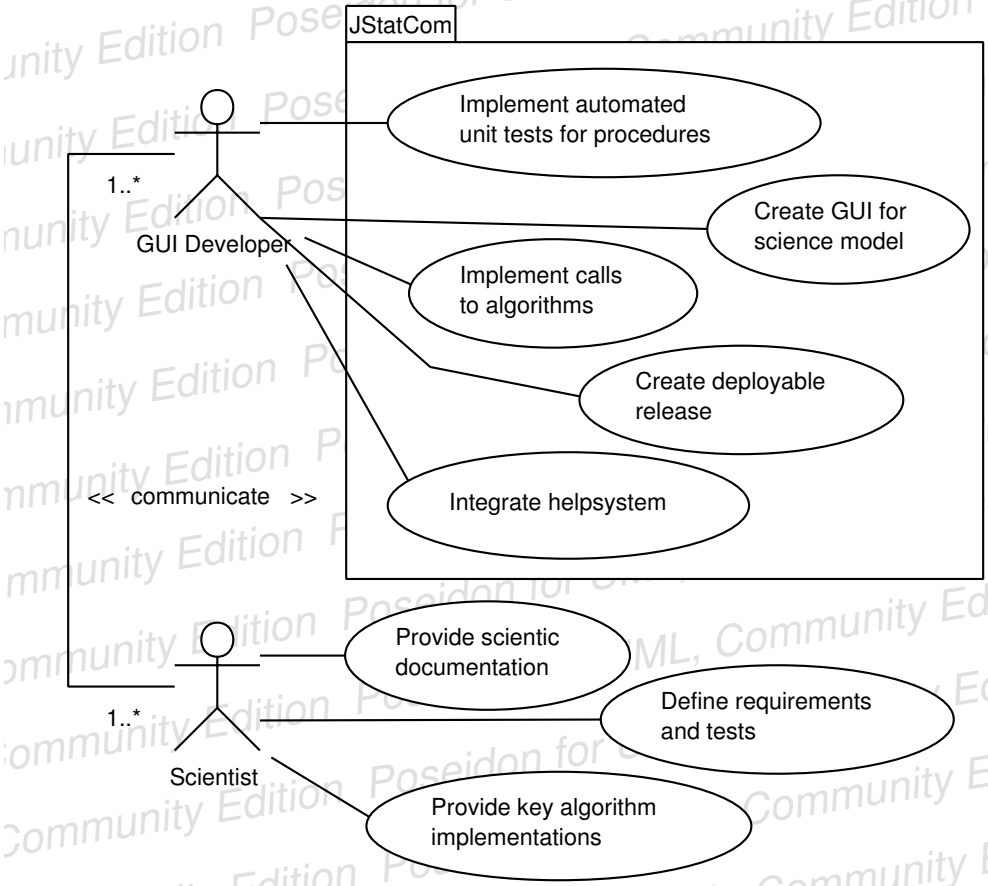

Fig. 2. Use cases for JStatCom

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. Thus it supports the Java developer in incorporating the procedures quickly, in laying out the GUI with specialized components, setting up a helpsystem and managing sets of automated unit tests.

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. [2] Algorithm implementations have to be provided as resources for the respective engine. For Ox these would be `.ox` or `.oxo` files. GUI devel-

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

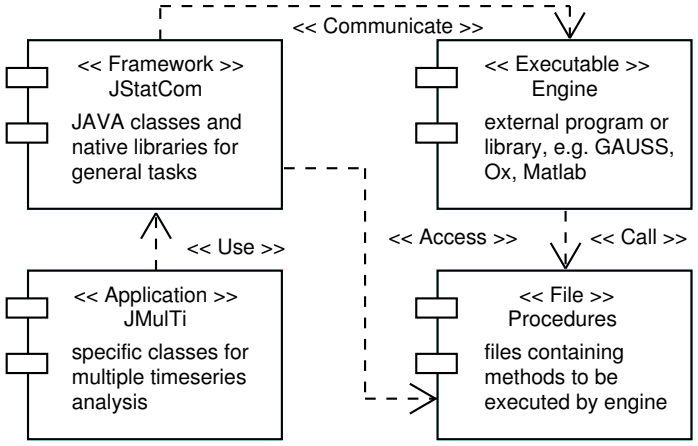

Fig. 3. Components of JStatCom

opers create the application, whereas scientists should provide the algorithm resources.

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 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 Type System, the Symbol Management and the Engine system are described in greater detail.

Element Name Implementation Units (`com.jstatcom`)	Element Responsibility

Data Model
`model`
`model.control`
`util`
	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`
`util`
	This module collects all classes that are especially designed for time series analysis. There are types to represent dates, date ranges and series, but also a number of specialized GUI components, like the time series selector.

Components
`component`
`table`
`equation`
`util`
	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, as well as input validating text fields for numbers, number ranges, dates and date ranges. The top level application frame is provided, as well as the module extension mechanism and the interface to the help system. The subsystem **Equation** is used to display GUI objects for models in matrix notation.

Parsers `parser`	This module contains generated parsers for the `TSCalc` language, for date expressions and for number ranges.

Engine
`engine`
`engine.gauss`
`engine.grte`
`engine.stub`
`engine.mlab`
`engine.ox`
	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**.[3] It also has the **PCall** system for procedure calls.

Table 1: Elements of JStatCom

3.1 Type System

JStatCom needs to represent data internally, because it maintains inputs and results of econometric 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, like for example

[3] The Stub engine can be used to call compiled dynamic link libraries directly from Java, without the need to write a dedicated wrapper library first.

Ox. 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 completely hidden from the developer and managed automatically by the engine implementations.

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 employed 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, which is different from the operation of most engines, including Ox. For example, a variable declared once can take `OX_STRING` and `OX_INT` values. The representing object changes its state accordingly. The benefit is, that the Ox programmer does not have to specify types explicitely, thus the syntax of the language is simplified. But the drawback is, that there is a lack of static checking. However, for JStatCom type-safety is a desired feature, because typically, attempts to change types of data objects after they have been created would be programming errors.

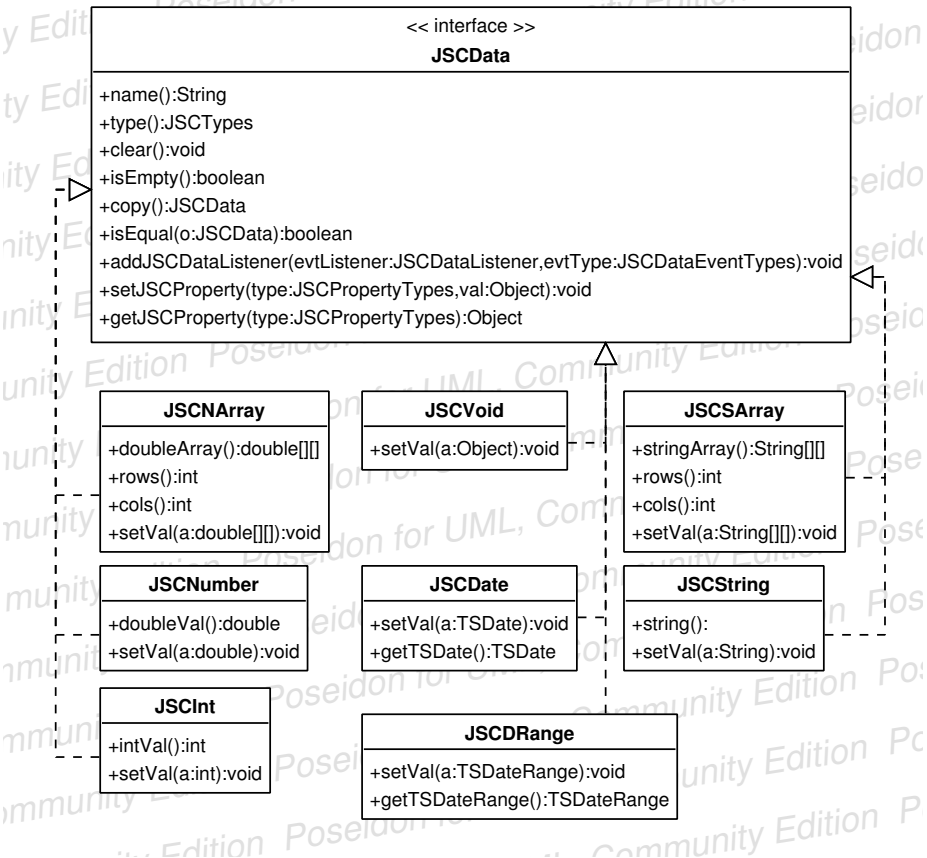

Fig. 4. Type System

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 choosen, 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 straighforward 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 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.

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 2 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 2: Type conversion between JStatCom and Ox

3.2 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 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.

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. 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 desireable 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.

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.

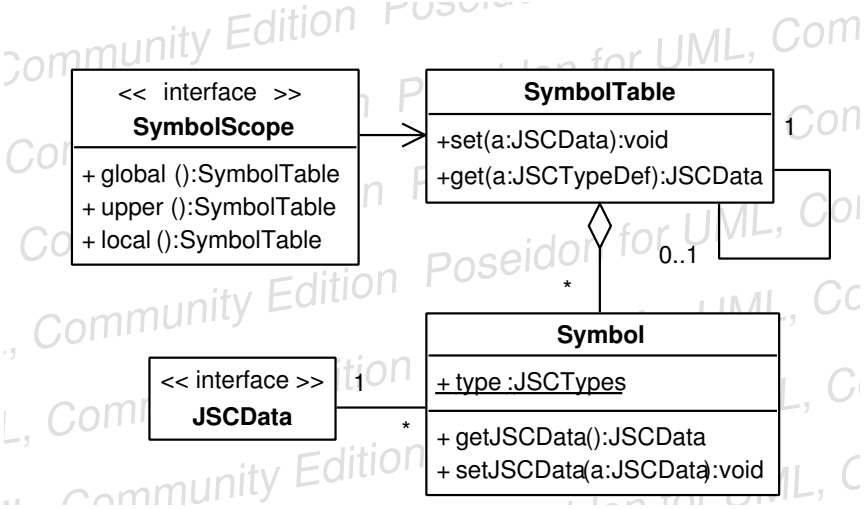

Fig. 5. Accessing shared data repositories

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:

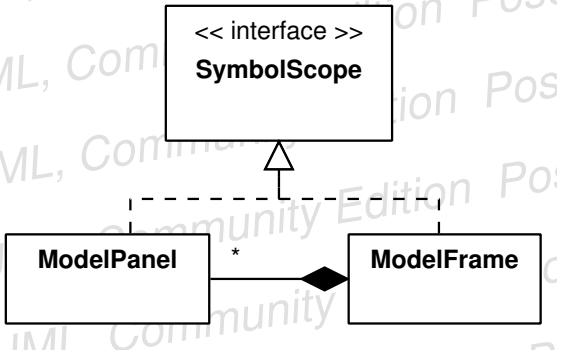

Fig. 6. SymbolScope inheritance

- `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

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. 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. 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 modeltypes 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.

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`, `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 diagnostic tests might go to `local21`. This data need not to

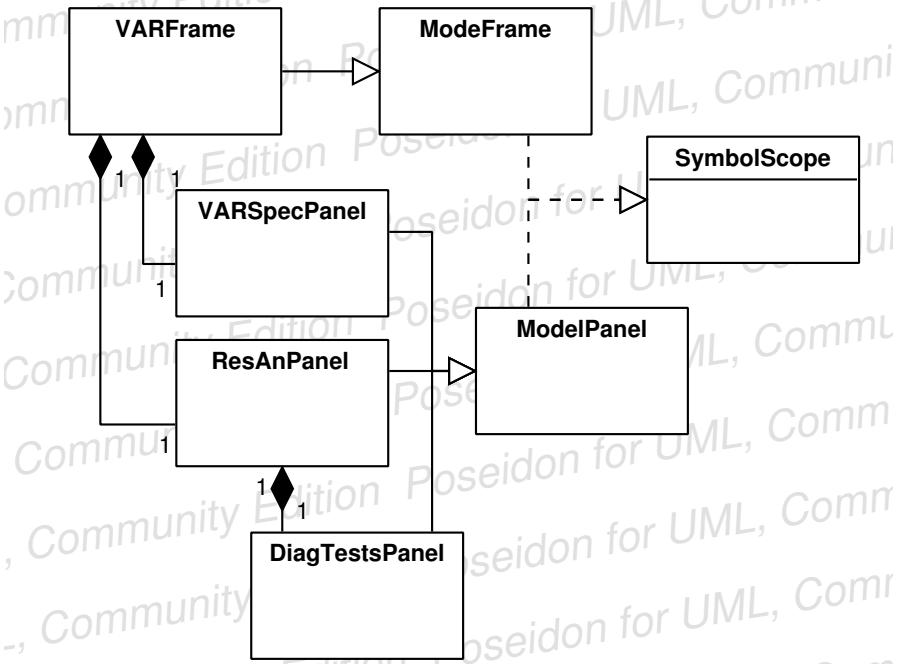

Fig. 7. Class structure of a hypothetical VAR frame

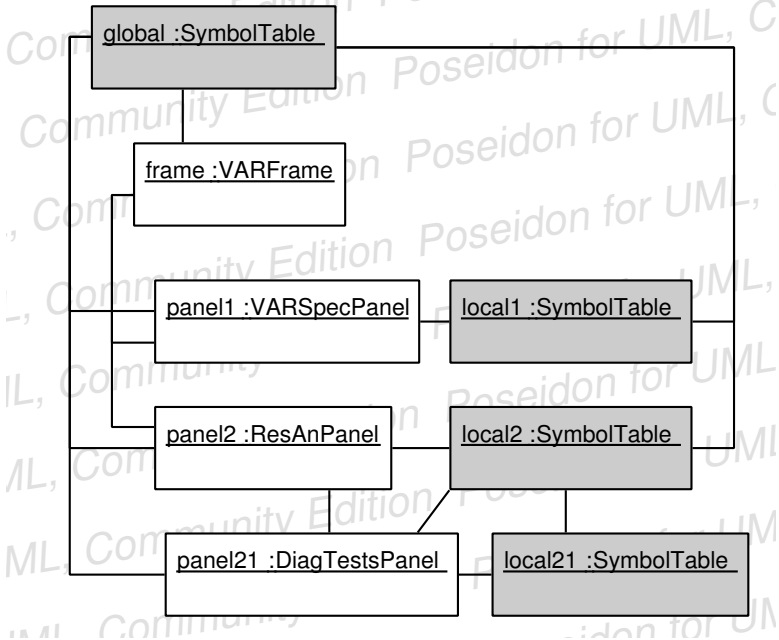

Fig. 8. Snapshot of model objects and shared data with different scopes

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, which is another subsystem of JStatCom that provides access to variables that are currently used. A desription 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 setEstimationMethod(){
    global().set(new JSCString("EstimationMethod", "OLS"));
  }
 } // end VARSpecPanel
```

```
// panel for residual analysis
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 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();
    ... // 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 extendable applications with JStatCom.

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 the Ox engine, the installation of Ox console is required together with the extra packages that are used. The inner workings of the Engine system are not described here, but rather how clients can use it.

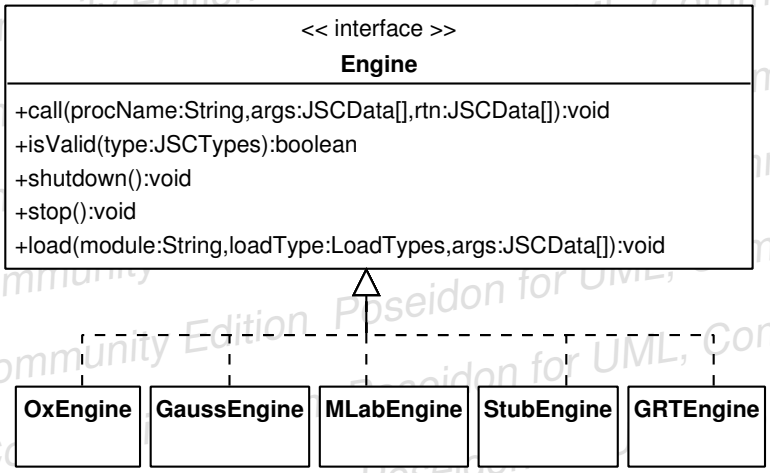

Fig. 9. Engine interface and available implementations

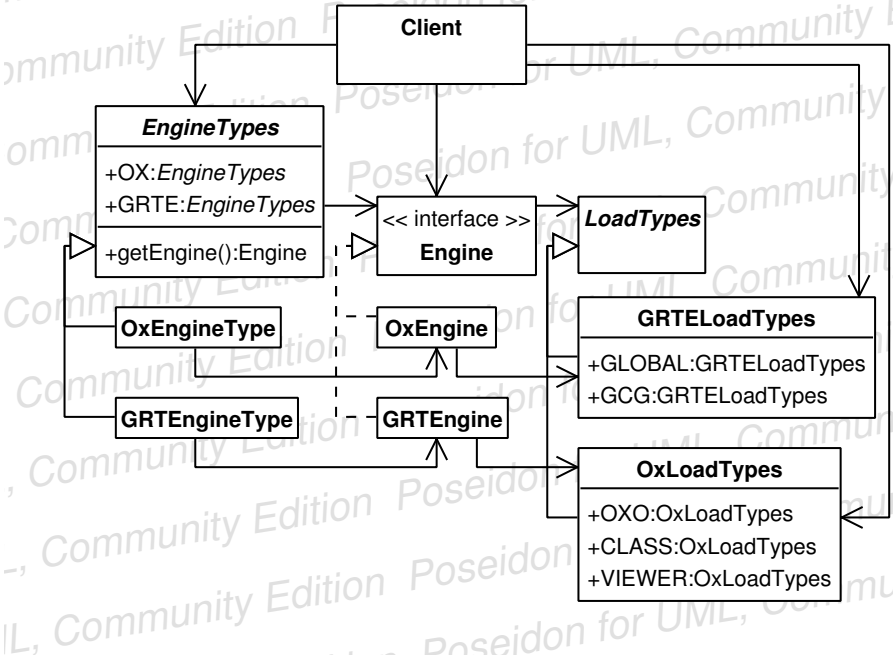

Fig. 10. Client using the Engine system

The framework tries to provide access to different engine implementations via a unified interface. Figure 9 presents the complete interface `Engine` and all implementations currently available. Clients should use the engine only via its abstract implementation, thus having the same calls for every implementation. 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 parametrized function `load` to address these issues. The method takes a parameter of type `LoadTypes` that defines the specific action to carry out.

Figure 10 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, Mathematica and Xplore. 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, the J/Link package for Mathematica or the MD*Crypt library for Xplore, to name just a few.

3.4 Calling the Ox Engine

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 the 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. The requirement to have a single class that provides methods that can be called from Java could actually be seen as an enforcement to use object-oriented programming, not only on the Java side, but also in Ox.

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.

3.5 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. Lim-

itations stem from the fact, that usually system specific dynamic link libraries are required. Those dlls might only be available to a certain operating system. 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 (Liang, 1999). 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.

3.6 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. A more elaborate real world Ox module will be presented in the next section.

```
#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 snipnet 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 an Ox resource. The Symbol Management is not involved here, because no data is shared. It will be used in the following real world example.

4 Programming with the Framework

After introducing the system JStatCom as a whole and describing some of its core elements, this section presents a more realistic development scenario. The example will show, how the Ox MSVAR package (Krolzig, 1998) could be used to create an application for Markov-Switching VAR analysis. However, for simplicity only the most basic features are implemented. But it should be straightforward to extend this example to turn it into a useful software. It should be mentioned that the MSVAR package is already implemented as a subclass of the Ox class `ModelBase`, such that it can be used with the GiveWin system, providing a graphical user interface. Therefore, this example can be used to compare the different implementations and the pros and cons of each system.

The JStatCom implementation gives more freedom to design the user interface and to combine the Ox package with other modules that are not directly related to it. One could even combine procedures written in Ox with algorithms implemented in other languages, such as Gauss or Matlab. The obvious drawback is that coding in Java is required, and that more code needs to be produced. In this respect, the Ox solution is, not surprisingly, simpler. On the other hand, one could customize JStatCom to create a subclass of `ModelFrame` which already has quite similar functionality to the `ModelBase` class. Instances of that new class could be configured by a relatively simple settings file, for example. This would be an extension of the framework for a specific problem domain.

4.1 Typical Steps

Creating applications with JStatCom consists of a number of steps that are always similar. Here it is assumed, that an Integrated Development Environment (IDE) is used. This is a software to support Java development projects. There is a number of tools available, some of them are Open-Source, like for example Eclipse. Development with an IDE is dramatically more efficient than using only a text editor and a compiler, especially when projects grow and when GUI layout is needed. However, it is by no means required to use such a tool, and a simple application could also be created "by hand". This text will not describe, how projects are set up with a specific software, however, there is detailed documentation on the web. The general steps in the development process are:

(1) download and unpack JStatCom
(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 your development tool

(3) create your top level component by subclassing `ModelFrame`, choose an appropriate name and title, and compile it (typically done automatically by IDE)

(4) put the fully qualified classname of your frame in the file `modules.xml`, for example add the line: `<Module class="msvar.MSVARFrame"/>`

(5) start your application with the batch file `app.bat`, maybe you need to add the classpath of your newly created class before, check it

(6) now go back to your development tool and implement all panels, use subclasses of `ModelPanel` if you need access to shared data

(7) implement calls to the external modules implementing the math algorithms, put all Ox modules that are used in the `jox` subdirectory, set up and run sets of automated unit tests for all engine calls

(8) repeat 5, 6 and 7 until all features are implemented, create a deployable version (for example, just zip the project directory)

One should also provide documentation for the user as a helpset. The add-on tool JHelpDev 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. Together with Latex2html or other converters, one could write the helpsystem completely in Latex, which is especially useful if many formulas are used. But details are omitted here.

In step 4 one puts the classname in a configuration file `modules.xml`. This way, the new frame will be recognized as a module when the JStatCom `TopFrame` is invoked. This is the true top level component of all model frames and it provides general functionality for importing data, transforming time series, accessing model frames and providing the helpsystem. The `TopFrame` is somewhat similar to GiveWin, but it can be configured in many ways. Some simple adjustments, like title, version, splashscreen and about-information can be set in the file `app.properties`. If further changes are wanted, one would have to subclass `TopFrame`. This way, almost everything can be changed in the behaviour and appearance of the application.

4.2 Creating a GUI

As already mentioned, the MSVAR Ox package is used together with JStatCom to create an application that is able to apply the provided methods on arbitrary datasets. However, the MSVAR package is very flexible and contains various modelling features. A GUI for it could become quite complex, therefore for this motivating example only a small subset of all features is implemented.

The MSVAR package comes with several example files, one of them is the following:

```
/**
 * MSM(2)-AR(4) Model of the US Business Cycle
 * see: Hamilton (1989), Econometrica 57, 357-384.
 * (c) Hans-Martin Krolzig, Oxford, 2002
 */
#include <oxstd.h>
#import  <msvar130>

main()
{
  decl msvar = new MSVAR();
  msvar->IsOxPack(FALSE);
  msvar->Load("gnp82.xls");
  msvar->Select(Y_VAR, {"DUSGNP", 0, 4});
  msvar->SetSample(1951,1,1984,4);
  msvar->SetModel(MSM, 2);
  format(120);
  msvar->Estimate();
  msvar->StdErr();
  msvar->PrintStdErr();
}
```

It 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. A GUI for this should provide an interface to load the data, set the sample, specify the number of AR lags and estimate the model. This way one could use the method to identify business cycles in different countries. All other possible variations, like changing the number of regimes, estimating a VAR, allowing for exogenous variables, and changing the type of the switching-regression, are omitted here for clarity. Adding them would be a straightforward extension of this example.

First the Ox wrapper module `msvarwrapper.ox` is given, providing an adapter class for the `OxEngine`. It should be noted, that one could also have used only the header of this file to load the `msvar` package into the workspace. In this case, an `MSVAR` object could have been created directly from Java. However, because the `Select` method takes a mixed type array as argument, it would not have been possible to call it directly from Java. Therefore the wrapper class is needed. It serves as an adapter between the `OxEngine` and the `MSVAR` class. `MSVARWrapper` is a subclass of `MSVAR` and inherits all methods from that class. It can be used instead of `MSVAR` with the same calling semantics. This is very handy and always possible, because Ox has no mechanism to restrict inheritance, like for example Java with the `final` keyword. From the

include statements it can be seen, that `msvar` and `gnudraw` are needed, without `gnudraw`, the module would not link when called from Java, because the graphics functions would not be found.

```
#include <oxstd.h>
#include <packages/gnudraw/gnudraw.h>
#import  <msvar130>

class MSVARWrapper:MSVAR  {
  MSVARWrapper();
  SelectY(const name, const startLag, const endLag);
  PrintSetup(const number, const fName);
}

MSVARWrapper::MSVARWrapper(){
  MSVAR();
}

MSVARWrapper::SelectY(const name, const startLag, const endLag){
  Select(Y_VAR,{name, startLag, endLag});
}

MSVARWrapper::PrintSetup(const number, const fName){
  format(number);
  fopen(fName, "l");
} // end of msvarwrapper.ox
```

One can see, that the problematic `Select` method is now interfaced by `SelectY`. The style of the adapter currently only allows to select a single variable, therefore only an AR model is allowed. For more variables, arrays of the respective names, start and end lags could be used as parameters and the selection array could be created via a loop. There should also be different select methods, like `SelectX` for exogenous variables, `SelectT` for the threshold and `SelectS` for the regimes. The method `PrintSetup` is needed to call static functions that are not members of a class, which are `format` and `fopen`.

Now the top level frame for the MSVAR analysis GUI module is given. It is a subclass of `ModelFrame`, although in this simple example a standard `JInternalFrame`, the superclass of `ModelFrame`, would have been sufficient. This is, because there is just one panel contained, and the global symbol table could therefore as well be the local table of the specification panel. However, using `ModelFrame` should be the default and has no recognizable performance implications.

```
package msvar;
import com.jstatcom.model.ModelFrame;

public class MSVARFrame extends ModelFrame {
  private msvar.MSVARSpecPanel msvarSpecPanel = null;
  public MSVARFrame() {
   super();
   initialize();
  }

  // init tasks, generated by IDE
  private void initialize() {
   this.setContentPane(getMSVARSpecPanel());
   this.setTitle("MSVAR Analysis");
   this.setSize(543, 505);
  }

  // adds specification panel, generated by IDE
  private msvar.MSVARSpecPanel getMSVARSpecPanel() {
   if (msvarSpecPanel == null) {
     msvarSpecPanel = new msvar.MSVARSpecPanel();
   }
   return msvarSpecPanel;
  }
} // end of MSVARFrame.java
```

The next code section contains the Java class `msvar.MSVARSpecPanel` with the specification panel and the Ox call. Here the main work is done, which is variable selection, lag input, estimation invocation and output presentation. This class uses several JStatCom components that are very helpful in that context, namely `TSSel` for variable selection, `ResultField` to present output and `NumSelector` to retrieve lag input that is validated against a predefined range. The use of these components is not described here in detail, this is left to the Java API documentation. The rest of the graphical components involved are standard Swing beans (Eckstein et al., 1998).

Although the code is pretty long, one has to note, that most of it can be generated with a visual layout tool, similar to Visual Basic. These code sections are marked with "generated by IDE". Such a tool should be part of the IDE software that is used. With a bit of experience, one can develop even complex GUIs in little time. Especially layout management needs a bit of care sometimes, but this is not discussed here.

```
package msvar;
import com.jstatcom.engine.*;
import com.jstatcom.engine.ox.*;
import com.jstatcom.io.*;
import com.jstatcom.model.*;
import com.jstatcom.ts.*;
import java.awt.event.*;
import java.io.*;

// Specification of MSVAR model.
public class MSVARSpecPanel extends ModelPanel {
 private com.jstatcom.ts.TSSel tsSel = null;
 private javax.swing.JButton jButton = null;
 private com.jstatcom.component.NumSelector numSelector = null;
 private javax.swing.JLabel jLabel = null;
 private com.jstatcom.component.ResultField resultField = null;
 private javax.swing.JPanel jPanel = null;

 // default constructor, generated by IDE
 public MSVARSpecPanel() {
  super();
  initialize();
 }
 // init method, generated by IDE
 private void initialize() {
  this.setLayout(new java.awt.BorderLayout());
  this.setSize(550, 503);
  this.add(getTSSel(), java.awt.BorderLayout.EAST);
  this.add(getJPanel(), java.awt.BorderLayout.CENTER);
 }
 // time series selector, generated by IDE
 private com.jstatcom.ts.TSSel getTSSel() {
  if (tsSel == null) {
   tsSel = new com.jstatcom.ts.TSSel();
   tsSel.setEndogenousDataName("Y_VAR");
   tsSel.setEndogenousStringsName("Y_NAMES");
   tsSel.setExogenousDataName("X_VAR");
   tsSel.setExogenousStringsName("X_NAMES");
   tsSel.setDateRangeName("MSVAR_DRANGE");
   tsSel.setOneEndogenousOnly(true);
 }
 return tsSel;
 }
 // estimate button with listener
 private javax.swing.JButton getJButton() {
   if (jButton == null) {
    jButton = new javax.swing.JButton();
```

```
jButton.setText("Estimate");
// connects action to estimate button
jButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // usage of PCall system, creates extra Thread
    PCall job = new PCall() {
     // mimics the main method of hamilton example
     public void runCode() {
      Engine ox = engine();
      ox.load("msvarwrapper", OxLoadTypes.OXO, null);
      ox.load("MSVARWrapper", OxLoadTypes.CLASS, null);
      ox.call("IsOxPack", new JSCData[]{
        new JSCInt("oxPack", false)}, null);
      TSDateRange range = global().getSymbol(
       "MSVAR_DRANGE").getJSCDRange().getTSDateRange();
      ox.call("Create", new JSCData[]{
       new JSCInt("freq", range.subPeriodicity()),
       new JSCInt("main_l", range.lowerBound().mainPeriod()),
       new JSCInt("period_l", range.lowerBound().subPeriod()),
       new JSCInt("main_u", range.upperBound().mainPeriod()),
       new JSCInt("period_u", range.upperBound().subPeriod())},
       null);
      JSCNArray data = global().getSymbol("Y_VAR")
                                .getJSCNArray();
      JSCSArray namesY = global().getSymbol("Y_NAMES")
                                .getJSCSArray();
      ox.call("Append", new JSCData[]{data, namesY}, null);
      ox.call("SelectY", new JSCData[]{
       new JSCSArray("aName", namesY.stringAt(0, 0)),
       new JSCInt("startLag", 0),
       new JSCInt("endLag", getNumSelector().getIntNumber())},
       null);
      ox.call("SetSample", new JSCData[]{
       new JSCInt("main_l", range.lowerBound().mainPeriod()),
       new JSCInt("period_l", range.lowerBound().subPeriod()),
       new JSCInt("main_u", range.upperBound().mainPeriod()),
       new JSCInt("period_u", range.upperBound().subPeriod())},
       null);
      ox.call("SetModel", new JSCData[]{new JSCInt("MSM", 5),
       new JSCInt("regimes", 2)}, null);
      String fName = JSCConstants.getSystemTemp()
                   + "/msvar.out";
      ox.call("PrintSetup", new JSCData[]{
       new JSCInt("width", 120),
       new JSCString("outFile", fName)}, null);
      ox.call("Estimate", null, null);
      ox.call("StdErr", null, null);
```

```
          ox.call("PrintStdErr", null, null);
          // appends output generated in Ox to output buffer
          output.append(FileSupport.readTextFile(fName) + "\n\n");
          new File(fName).delete();
         }
         public Engine engine() {
          return EngineTypes.OX.getEngine();
         }
        };
      job.setName("MSVAR Call");
      // sets result field as output component
      job.setOutHolder(getResultField());
      // queues the job in the background thread
      job.execute();
     }
    });
 }
 return jButton;
}
// number range, generated by IDE
private com.jstatcom.component.NumSelector getNumSelector() {
 if (numSelector == null) {
   numSelector = new com.jstatcom.component.NumSelector();
   numSelector.setRangeExpr("[0, 10]");
 }
 return numSelector;
}

// lags label, generated by IDE
private javax.swing.JLabel getJLabel() {
 if (jLabel == null) {
   jLabel = new javax.swing.JLabel();
   jLabel.setText("Endogenous lags");
 }
 return jLabel;
}
// result field, generated by IDE
private com.jstatcom.component.ResultField getResultField() {
 if (resultField == null) {
   resultField = new com.jstatcom.component.ResultField();
 }
 return resultField;
}
// add all components, generated by IDE
private javax.swing.JPanel getJPanel() {
 if (jPanel == null) {
   // constraints for layout omitted, generated automatically
```

```
    jPanel.add(getResultField(), consGridBagConstraints1);
    jPanel.add(getNumSelector(), consGridBagConstraints2);
    jPanel.add(getJLabel(), consGridBagConstraints3);
    jPanel.add(getJButton(), consGridBagConstraints4);
   }
   return jPanel;
 }
}
```

The example code contains the method `getTSSel` to initialize the time series selector. One can set the names of the variables that are selected. They will be stored in the global symbol table under the respective names. The `getNumSelector` method sets up the number selector with the interval $[0, 10]$. Whenever input validation fails, a dialog is shown and the value is set back to the previous one.

But the most interesting method to look at is `getJButton`. Here the estimate button is configured and an action listener is attached. This listener invokes the call to the engine and mimics the behaviour of the `main` method in the MSVAR example file. There is the PCall system involved here, which is used to handle procedure calls. It is not mandatory to use this system, but it has many advantages. One of the major benefits is, that it can execute the call in a new thread. This way the GUI is still reactive, even if a lengthy computation is running. There is not a new thread for every `PCall` invocation, but instead a background thread is used, and new calls are queued until the previous call has finished. By calling the `PrintSetup` method of the Ox adapter class, the output is redirected to a file, which is known by the Java side. The contents of this file are then appended to the output buffer. The PCall system sets the contents of this buffer automatically to the output holder, which is the result field in this case. This way, text formatting on the Java side is not necessary.

4.3 Critique of the Implementation

Although the example reflects a real world situation and could serve as a starting point for a full featured analysis module, there is much potential to improve the presented implementation. First, the names of the global variables are just given as strings where they are needed. If one chooses to change the name of a variable, this would have an impact on all parts of the code where the variable is used. Especially if there are many variables to be shared, an extra class with the type definitions is worth being considered. Type definitions should not be implemented as plain strings, but rather as instances of `JSCTypeDef`. An example class could be:

```
final class MSVARConstants {
  public static final JSCTypeDef Y_VAR = new JSCTypeDef(
      "Y_VAR", JSCTypes.NARRAY,
      "The selected endogenous variables, no lag truncation.");
  public static final JSCTypeDef Y_NAMES = new JSCTypeDef(
      "Y_NAMES", JSCTypes.SARRAY,
      "The names of the selected endogenous variables.");
  ...
}
```

Instead of using strings, one should use these type definitions to reference variables. A side effect is, that the descriptions are put in the Symbol Control system when the symbol is referenced with this definition for the first time:

```
  // initialization in getTSSel
  tsSel.setEndogenousDataName(MSVARConstants.Y_VAR.name);

  // reference to endogenous variables
  global().get(MSVARConstants.Y_VAR).getJSCNArray();
```

Another obvious drawback is the nesting of the estimation call within the `MSVARSpecPanel`. It is much better to separate the procedure call in an extra class, which is an example of the *Command* pattern. This has the advantage, that calling logic and GUI code would be better separated. One could use the new class not only in one place, but it could be reused internally. This could be useful for a command implementing a call to display the autocorrelation function, for example. Another big advantage would be, that the class could also be created and checked with the help of automated unit tests, an invaluable tool to improve the quality of code that is under constant change. Any input parameters would have to be set in the constructor, the code might then be:

```
public final class MSVAREstCall extends PCall {
    private JSCNArray data;
    private JSCSArray namesY;
    private int lags;
    private TSDateRange range;
    private String fName = JSCConstants.getSystemTemp()
                           + "/msvar.out";

    public MSVAREstCall(JSCNArray y, JSCSarray names,
                        int arLags, TSDateRange range){
     setName("MSVAR Call");
     // maybe check inputs and copy values here
     this.yDat = y;
     this.nam = names;
     this.lags = arLags;
```

```
     this.range = range;
    }
    public void runCode() {
     Engine ox = engine();
     ox.load("msvarwrapper", OxLoadTypes.OXO, null);
     ox.load("MSVARWrapper", OxLoadTypes.CLASS, null);
     ox.call("Create", new JSCData[]{
      new JSCInt("freq", range.subPeriodicity()),
      new JSCInt("main_l", range.lowerBound().mainPeriod()),
      new JSCInt("period_l", range.lowerBound().subPeriod()),
      new JSCInt("main_u", range.upperBound().mainPeriod()),
      new JSCInt("period_u", range.upperBound().subPeriod())},
      null);
     ox.call("Append", new JSCData[]{data, namesY}, null);
     ox.call("SelectY", new JSCData[]{
      new JSCSArray("aName", namesY.stringAt(0, 0)),
      new JSCInt("startLag", 0),
      new JSCInt("endLag", lags)}, null);
     ox.call("SetSample", new JSCData[]{
      new JSCInt("main_l", range.lowerBound().mainPeriod()),
      new JSCInt("period_l", range.lowerBound().subPeriod()),
      new JSCInt("main_u", range.upperBound().mainPeriod()),
      new JSCInt("period_u", range.upperBound().subPeriod())},
      null);
     ox.call("SetModel", new JSCData[]{new JSCInt("MSM", 5),
      new JSCInt("regimes", 2)}, null);
     ox.call("PrintSetup", new JSCData[]{
      new JSCInt("width", 120),
      new JSCString("outFile", fName)}, null);
     ox.call("Estimate", null, null);
     ox.call("StdErr", null, null);
     ox.call("PrintStdErr", null, null);
   }
   public void finalCode(){
     // appends output generated in Ox to output buffer
     output.append(FileSupport.readTextFile(fName) + "\n\n");
     new File(fName).delete();
   }
   public Engine engine() {
     return EngineTypes.OX.getEngine();
   }
};
```

Currently this class only prints the results in text form. It would be desirable to read residuals, coefficient estimates, standard deviations, etc. back to Java. This can easily be implemented by just calling the respective methods of the `MSVAR` class. One could also factor some of the calls to the Ox wrapper class to

reduce the number of calls that are necessary from the Java side. For example, the creation of a database, appending variables and setting the sample could all be done within the `SelectY` method. The code required in Java would become much shorter.

These final remarks indicate some of the design guidelines that can be used for applications based on JStatCom. The idea is to have a set of best practices that ensure a high quality class design, which can easily be maintained, extended and tested.

5 Conclusion

It was shown, how the software framework JStatCom could be used to create graphical user interfaces for econometric routines with the help of Ox. The approach was demonstrated for an example application, which could well be extended to establish a useful system. Various subsystems of JStatCom address common problems that are inherent when designing graphical user interfaces for complex mathematical algorithms. A strong emphasis was put on the extendability and flexibility of the framework. It is hoped, that the presented approach is still simple enough to serve as a development platform for researchers in the field, who want to make available their algorithms to interested users.

References

Ashworth, M., Allan, R., Mller, 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-Universitt zu Berlin.

Bianchi, A., Caivano, D., Lanubile, F. and Visaggio, G. (2001). Evaluating software degradation through entropy, *Proc. 7th IEEE International Software Metrics Symposium*, London, pp. 210–219.

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.

Booch, G., Rumbaugh, J. and Jacobsen, I. (1999). *The Unified Modeling Language User Guide*, Addison-Wesley.

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.

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.

Eckstein, R., Lay, M. and Wood, D. (1998). *JAVA Swing*, O'Reilly.

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

Hamilton, J. D. (1989). A New Approach to the Economic Analysis of Nonstationary Time Series and the Business Cycle, *Econometrica* **57**(2): 357–384.

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

Krolzig, H.-M. (1998). Econometric Modelling of Markov-Switching Vector Autoregressions using MSVAR for Ox. Department of Economics, University of Oxford.

Liang, S. (1999). *Java Native Interface*, Addison-Wesley.

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.