



CHARLES UNIVERSITY, PRAGUE

DISTRIBUTED SYSTEMS RESEARCH GROUP

SOFA NetBeans Module

an introductory guide

Revision 1.0

June 2003

Contents

1	MODULE'S ESSENTIALS	3
1.1	Introduction	3
1.1.1	License Conditions	3
1.1.2	Availability	3
1.1.3	Supported Versions of the Netbeans IDE	4
1.2	Structure of the Module	4
1.2.1	SOFA TIR Browser	4
1.2.2	SOFA CDL Compiler	6
1.2.3	Visual Designer	7
1.2.4	Options for the Module	8
2	THE MODULE IN ACTION	10
2.1	Installing the Module	10
2.2	A Sample Example	13
2.3	Designing Components	14
2.4	Compiling CDL Files	21
2.5	Using the SOFA TIR Browser	22

2.5.1	Transactional nature of TIR	25
3	TROUBLESHOOTING	27
3.1	F.A.Q.	27
3.1.1	Security policy not set	27
3.1.2	Profile locked	27

Chapter 1

MODULE'S ESSENTIALS

1.1 Introduction

The purpose of this document is to provide basic information about the SOFANBModule, which is a plug-in module to the [NetBeans](#) integrated development environment. The module was developed by the [Distributed Systems Research Group](#)(DSRG) at [Charles University, Prague](#) to provide Java developers a tool for working with the [SOFA component model](#) the DSRG members have developed.

1.1.1 License Conditions

The module is distributed under the [GNU Lesser General Public License](#).

1.1.2 Availability

The module can be downloaded from the [ObjectWeb Forge](#). It is distributed as a standard *.nbm file ready to be installed into the NetBeans IDE, see section [2.1](#)

1.1.3 Supported Versions of the Netbeans IDE

The described module supports NetBeans versions 3.4.1 and the beta, RC1 and final releases of the version 3.5. These versions were current at the time of the last revision of the module. We anticipate that the module will be compatible with all further versions until the NetBeans version 4, which is expected to bring major changes into the OpenAPI used for writing NetBeans plugins.

1.2 Structure of the Module

Three main parts can be recognized in the module: a [TIR browser](#), a [compiler of CDL files](#) and a [tool for visual creation](#) of SOFA components. However, these parts are not independent — in fact, the most common use case of the module is that first a component is visually designed using the Visual Designer, subsequently it is compiled using the CDL compiler that outputs results of compilations into the TIR and finally the Java mappings can be generated calling the particular action in the TIR browser's context menu.

1.2.1 SOFA TIR Browser

The TIR (Type Interface Repository) browser can be used independently on the rest of the module's functionality. The SOFA TIR is a standard Java RMI server that uses RMI registry as a naming service and thus can be run anywhere on the Internet. After connecting to it, the TIR browser allows users to examine the TIR's contents. In the case of SOFA, the TIR contains types of the CDL language including their versions and other properties. From the browsable types, *modules*, *interfaces*, *interface operations*, *template frames*, *template architectures* and *providers* should be

mentioned.

Except browsing the TIR contents, several other functions of the TIR browser can be itemized:

- opening TIR in order to be modified by the CDL compiler output, and closing it by committing or aborting the performed operations
- generating component builders and java types from the information stored in the TIR
- creating new profiles
- inspecting properties of the types

The TIR has two operational modes: it can be set to a read-only mode or to a read-write mode. The read-only mode is default and means no TIR changes can be performed. Opening the TIR for writing means starting a new transaction. During the time the TIR is in read-write mode, its content is allowed to be modified (e.g. by the output of the CDL compiler) and all modifications are tracked. Eventually, the TIR must be explicitly set to the read-only mode either by committing or aborting. Committing the changes results in the definitive writing of the performed changes down into the TIR, aborting them results has such an effect if the TIR was not actually open.

Java mappings can be generated only when the TIR is set to read-only mode (i.e. its content is committed). The code generation is based on architectures — from any architecture a component builder is created together with mappings of all types that are used in its definition (especially interfaces and their elements) into a valid Java code.

New profiles can be created in both TIR states (read-only and read-write), however if they are created when the TIR is in the

read-write mode and subsequently the TIR is set to the read-only mode by aborting, such profiles are deleted during the abort.

1.2.2 SOFA CDL Compiler

The SOFA CDL compiler is used to compile CDL source files. The CDL stands for *Component Description Language* and it defines the way components are described in the SOFA Component Model. The definitions originate from the CORBA IDL language but a support for defining components is added.

There are three ways to obtain a CDL source file: It is possible to

- create a file with component definitions externally and open it via standard NetBeans menus
- use the template wizard to create and open an empty CDL file and write component descriptions directly in the NetBeans IDE
- use the template wizard to open a visual designer, create the CDL file visually and generate a CDL source file from it

If the compilation fails, the TIR is automatically set to the read-only mode aborting all operations and needs to be re-opened in order to compile again.

If the compilation successfully passes, the output is written to the TIR and then the TIR is used to do further processing (as described in operations the previous section). Especially it is necessary for the user to set the TIR to the read-only mode deciding to commit the changes or abort them.

1.2.3 Visual Designer

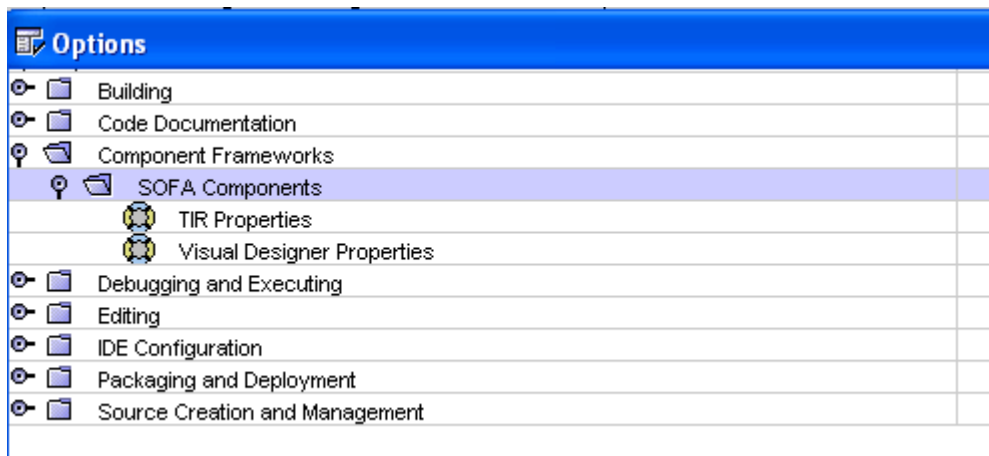
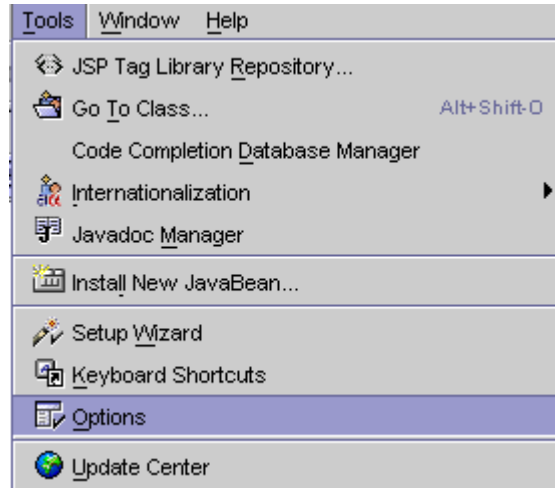
Visual designer is a tool that helps with creating SOFA Components. The main capabilities of the designer include:

- defining interfaces, template frames and primitive architectures including interface and frame protocols
- creating compound architectures (i.e. first-level structures of) of single components (interconnections between interfaces of subcomponents)
- encompassing the definitions by modules
- creating hierarchies of composed components
- specifying user-code if necessary
- saving a visually designed component in a file (with `.sofa` extension)
- generating a textual ready-to-compile CDL source file into the NetBeans IDE from the visually design

The result is usually converted into a common CDL source file and opened in a NetBeans IDE window, where all actions as described in the previous sections can be applied.

The sketched components can be saved on disk by the Visual Designer (files with the `*.sofa` extension) in order to be re-opened and further elaborated later in the Visual Designer. These files are created by making some internal objects persistent, therefore it can be used outside the Visual Designer. There is currently no support for exporting the visually designed components into standard graphics files.

1.2.4 Options for the Module



For users' convenience, a possibility of a customization of the module via standard NetBeans system options has been added — choosing Tools -> Options in the NetBeans IDE pull-down menu, the options window appears and subsequently the item Component Frameworks -> SOFA Components contains two sub-options: one for setting the TIR browser and compiler, the other for setting the Visual designer.

The TIR Properties contain the following items:

gencodedir	C:\SOFA\Components
global	False
profile	Default
testprotocols	False
tirhost	localhost
tirport	2000
useprotocols	False

gencodedir — a directory into which the java files (mappings of types and component builders) will be generated; implicitly, the user's home directory or a working directory is set

profile — what profile from the profiles in the TIR will be chosen; if an invalid profile is specified, an exception will be thrown

tirhost — a host on which the TIR runs; default is the localhost

tirport — a port on which the TIR listens; default is 2000

useprotocols — whether the CDL compiler should take protocols specified in the file into consideration and test their syntax validity; if not, the compiler ignores parts where protocols are specified and they are not reflected in the TIR.

testprotocols — whether the CDL compiler should test conformance between protocols (interface-interface and interface-frame)

The Visual Designer Properties contain the following items:

Default module	SOFA
Default provider	CUNI::SOFA

Default module — specifies into which module the initial frame (and interfaces) will be placed after opening the visual designer; individual modules are separated by the double-colon

Default provider — specifies into which provider the initial architecture will be placed after opening the visual designer

Chapter 2

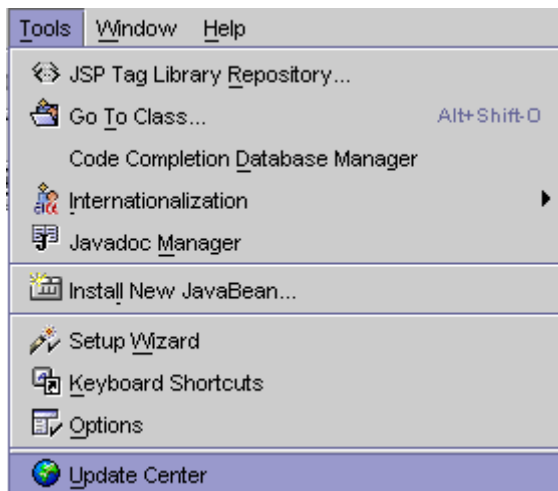
THE MODULE IN ACTION

In this chapter, the SOFA NetBeans module step-by-step will be presented. This will be shown on a practical level — using an case study showing a typical task of designing and compiling a several components components and generating Java code based on those components. But before the module can be used, it must be downloaded and installed into the NetBeans IDE.

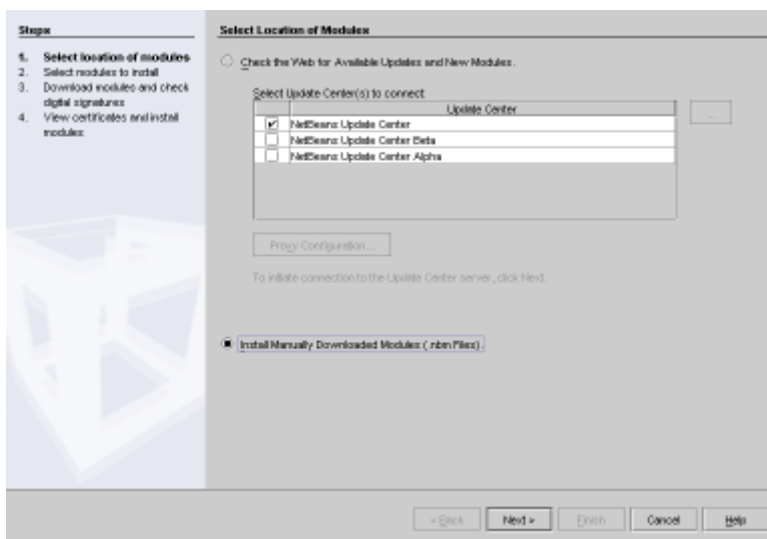
2.1 Installing the Module

Having downloaded the module (the `sofa.nbm` file) [from the ObjectWeb Forge](#), it can be installed into the NetBeans IDE. You can proceed in these steps:

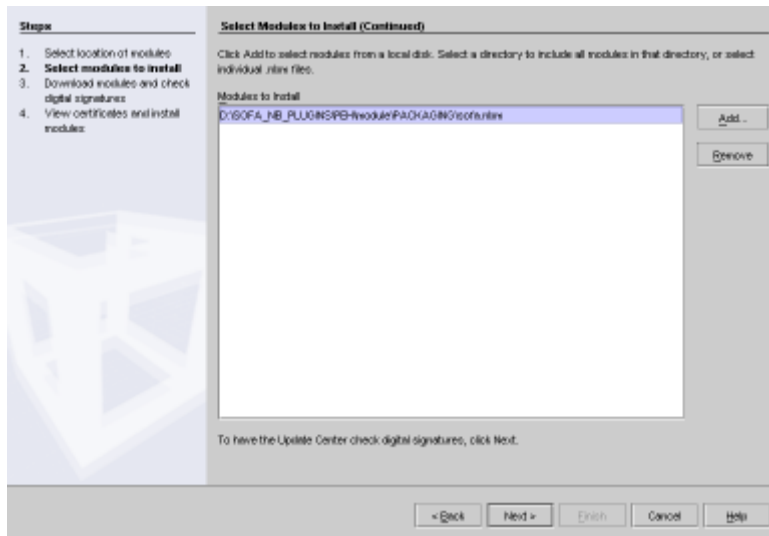
- (1) Click on NetBeans pull-down menu *Tools -> Update Center*



(2) Click on *Install manually ...*

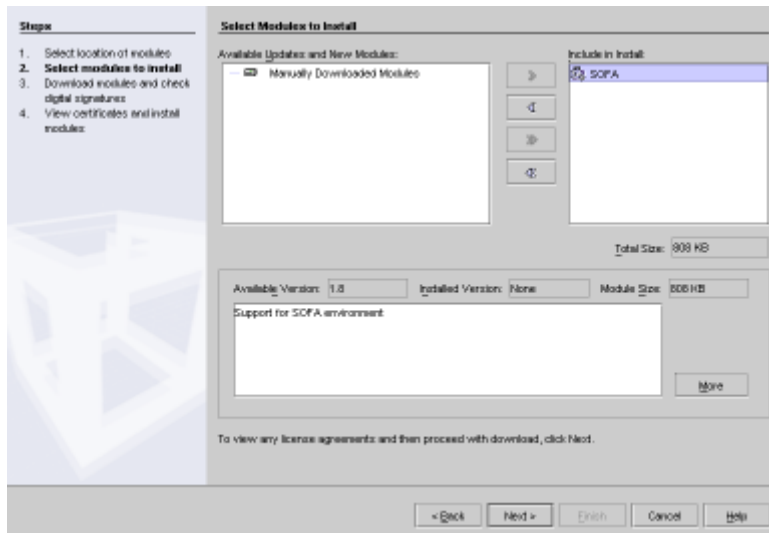


(3) Click on *Add..* and load the downloaded file



(4) Click on *Next*

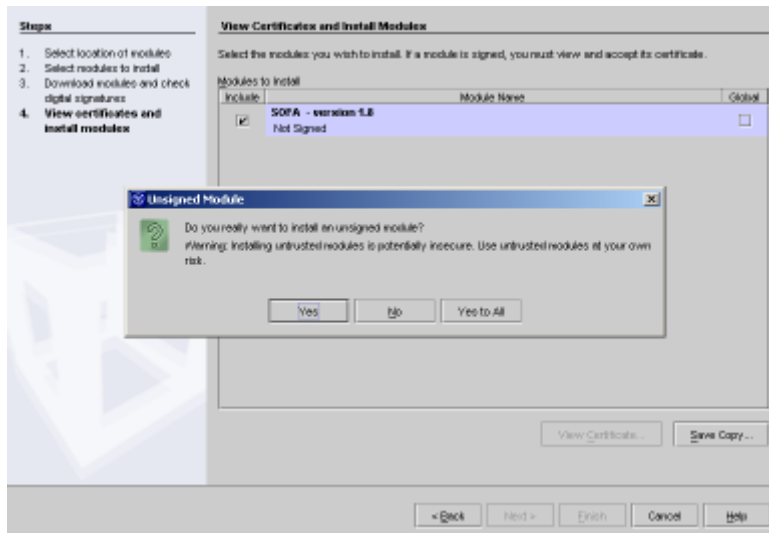
(5) In the new window, click on the SOFA module in the Include in Install textfield



(6) Then click on *Next*

(7) After you accept the license conditions, the module will be loaded to the NetBeans environment ready to install

(8) Check the *SOFA version 1.8* checkbox and accept installing the unsigned module



(9) Click on *Finish* and *OK* in the dialog box asking whether to restart the IDE. After that, the module has been installed.

2.2 A Sample Example

Suppose you have to create a simple component application for logging events. Those logs should include timestamps of the events and the given date format can differ in various countries according to local habits.

Having analyzed the problem, you have decided to use the following components:

FDateTimeStamp — a component that provides a single interface *IDateTimeStamp* that contains operations for various date formats; its architecture will be primitive (i.e. directly

implemented in Java)

FTimeLog — a component that provides logging capabilities (interface *ILog*) and requires the time formats in the interface *IDateTimeStamp*; its architecture will be primitive

FLog — a component that provides general logging capabilities (and requires nothing). In fact, it will be composed of the subcomponents of the previous two component types while delegating the logging capability to the subcomponent

FLogUtilizer — a component that may generate some events and requires the logging (the *ILog* interface)

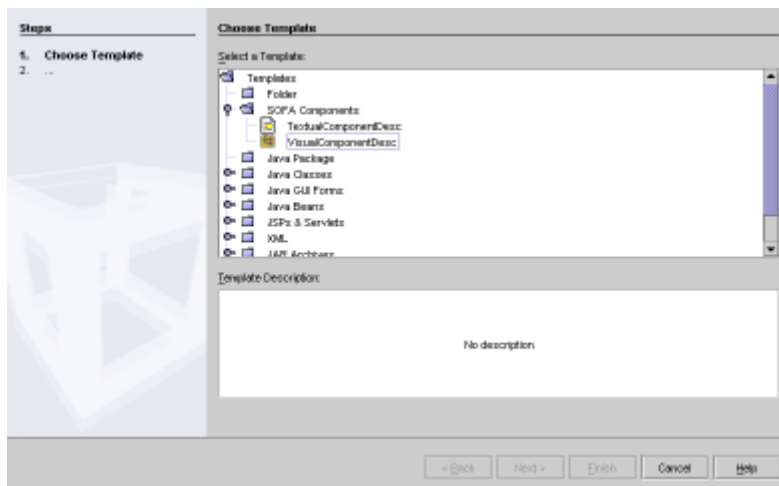
ATimeLogApp — a system component architecture that has no provisions or requirements (it implements *::SOFA::libs::Application*) but instantiates the *FLog* and *FLogUtilizer* components as instances.

2.3 Designing Components

Before you begin to design the components, you have to choose into which initial module respective provider the component interfaces and frames respective architectures will be placed. This is done via the system options as described in [section 1.2.4](#).

For working with SOFA components, a new NetBeans workspace has been created by the module. The Visual Designer will create components in docked windows within this workspace and whenever the **.sofa* modules will be loaded. You are encouraged to place also the **.cdl* files with the CDL source into this workspace.

Provided you have already mounted a directory the files should be stored in, click on *File->New* from the main menu and from the offer of templates choose *SOFA Components -> VisualComponentDesc*. See picture below.



Fill the name *LoggingSofaApp* into the field that appears (in general, you can leave it default) and click *Finish*. If the SOFA workspace was not active, it will become active. You can see the workspace is divided into three parts

Explorer window — displays the content (files) of mounted locations; each element (node) is a file: clicking on nodes, you can open chosen files in the editor window

Properties window — displays various properties of a file that is selected at the given time as an active node; for the Visual Designer there is quite a lot of tabs with properties that will be discussed later.

Editor window — displays the content of the node selected in the Explorer window; for the Visual Designer, it is a docked window with the actual graphical representation of components; initially an empty default frame (called *Frame1*) will be shown.

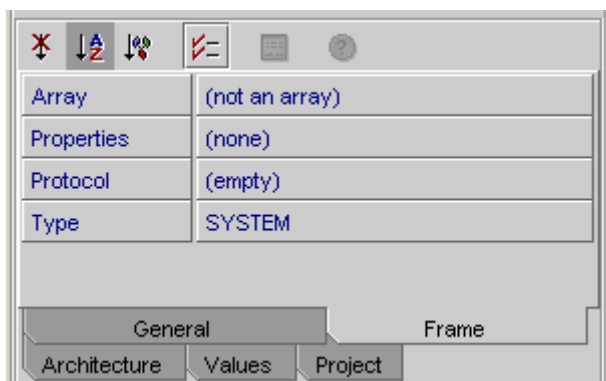
The visual component designer follows the top-down approach for component creation. First, highest level architecture is designed. The highest level architecture can implement two types of frames:

An ordinary frame — this case is used when the designed component is supposed to be included in other components

The system frame — this case is used when the component will be an application component; its frame is empty because this is the highest level component encapsulating lower level components and its frame has no interfaces (no delegation and subsumption is used)

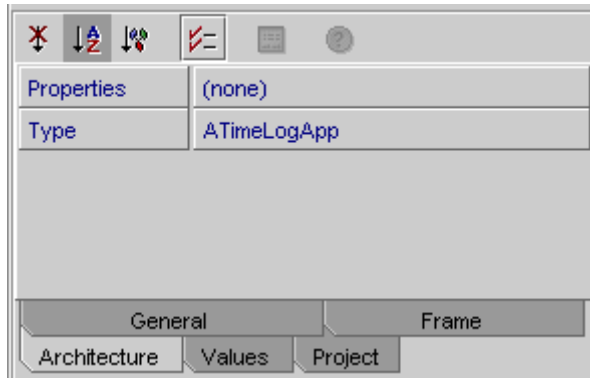
In our example, the latter is the case. To create an application component, you have to use a special name *SYSTEM* for the application frame. This is important for generating a correct CDL code for it forces the CDL-generator to leave the frame out and generate the application architecture implementing the system *SOFA::Libs::Application* frame.


The frame name can be set in the *Frame* tab in the Properties window (under the name *Type*). Along with frame type, you can set the following attributes: frame protocol, properties (which are kind of component parameterization that can be done from deployment descriptor) and settings for arrays. In our case, the type will be set to *SYSTEM*, all other fields will be left blank. Protocol is not specified for there is no communication on the system frame.



As the next step, you need to set the name of the designed ar-

chitecture. Therefore switch to the *Architecture* tab and fill the *Type* field with *ATimeLogApp*. The other field (*Properties*) leave blank.



We want the application architecture to have two subcomponents: a component that provides logging capabilities and a component that requires them. In order to do that, go to the editor window and pick the second button from the left¹  and put two subcomponents on the *SYSTEM* frame². Click on the particular subcomponent, the Properties window will appropriately change to reflect attributes of the given subcomponents. Set individual properties similarly as in the previous case (i.e. frame names *FLog* resp. *FLogUtilizer*) and architecture names *ALogUtilizer* resp. *ALogWithTimeStamp*. Set also the frame protocols: `?iLog.doLog*` resp. `!iLog.doLog*`



Besides, in the *General* tab set names of the subcomponents to *scLog* and *scLogUtilizer*³

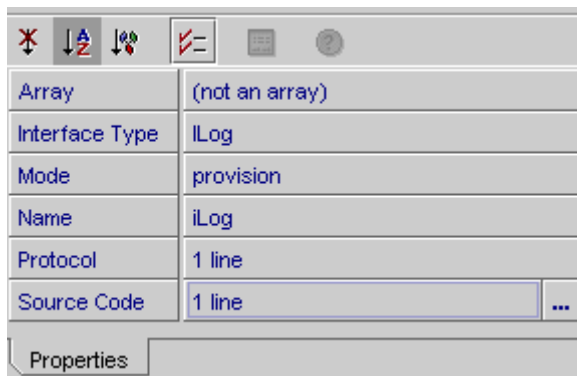
These two subcomponents are supposed to communicate via an interface dedicated for logging. Therefore click on the provided


¹If you cannot understand the meaning of individual buttons in the toolbar from their icons, leave the cursor on the particular button a bit longer and a descriptive hint appears

²Please, do not be confused by this; the subcomponents are, of course, put to the architecture

³Note that for an easy reading we use the following naming conventions: the interface, frame and architecture types commence with their initial letter in capital (i.e. "I" resp. "F" resp. "A") and instances with their non-capital initial letter; the "sc" means a subcomponent instance

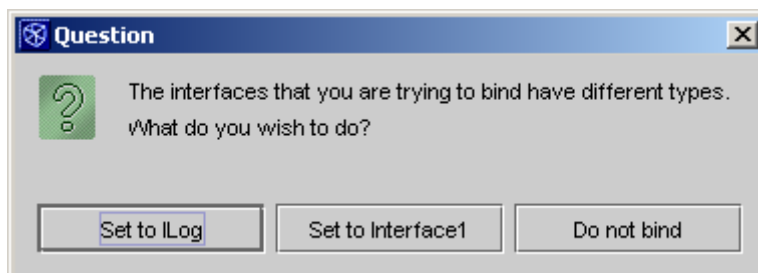
interface button (the third from left)  and place it onto the scLog subcomponent and on the required interface button (the fourth from left)  and place it onto the *scLogUtilizer* subcomponent. Now, click on one of the interfaces and the properties windows changes to reflect it. Fill the type name *ILog*, the instance name *iLog*. The bottom two properties allow you to more closely specify the interfaces. After clicking on them, a "..." button appears. Clicking on it, a new window appears where interface protocol resp. interface structure (methods, ...) can be specified. Content of these fields will be copied into appropriate parts of the generated CDL code ⁴. In our case, the source code is `void doLog(in string EventNumber);` and the protocol is `doLog*`



In general, you should do the same actions with the other interface too. However, since there is a restriction that only interfaces with of the same type can be bound in SOFA, do not bother with filling information for the other interface and press the button for creating ties (the sixth from left)  from the toolbar right away and click on one of the interfaces and then click on the other. The visual designer recognizes that the two interfaces are not the same and offers you to select whether to adjust one of the interface to the other or not to create the tie. Choose *Set to ILog* and the

⁴Without additional processing, i.e. no syntax checks will be done!!!

other interface will be appropriately adjusted (except the interface instance name which can be arbitrary, so you are encouraged to set its name to *iLog* manually).

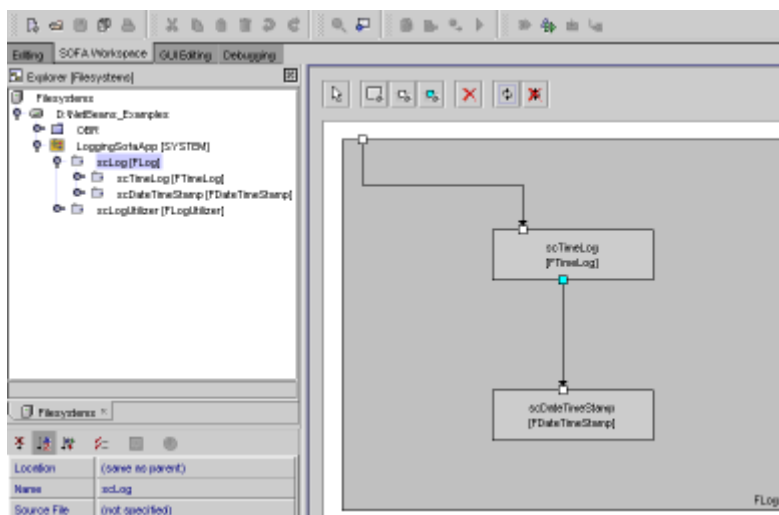


Now, the highest level of the component hierarchy is done. So we can start creating architectures of subcomponents. In our case, the architecture of *FLogUtilizer* will be implemented by the underlying implementation language and therefore the *scLogUtilizer* subcomponent will not be further elaborated (it will be reflected by the generated CDL code by the "primitive" keyword in the architecture specification). However, the subcomponent *scLog* will have a compound architecture that will contain components for working with time stamps⁵. In order to define the compound architecture of the subcomponent, double-click on the subcomponent (or alternatively you can unfold the *LoggingSofaApp* node in the Explorer window and double-click on the *scLog* node). The editor window now displays the subcomponent as the main panel onto which you can put subcomponents similarly as in the previous level. The Properties window will appropriately change as well. Now put two subcomponents onto the architecture of *scLog* and name them appropriately (*scDateTimeStamp*, *FDateTimeStamp* and *ADateTimeStamp* resp. *scTimeLog*, *FTimeLog* and *ATimeLog*).

Now you need to specify the frames by specifying their interfaces and creating ties. There will be one new interface: *IDateTimeStampFormats* that *scTimeLog* component to ask the *scDateTimeS-*

⁵The architecture was not named just "ALog" but "ALogWithTimeStamp" to better reflect this fact

tamp component about various formats of date-time stamps. Its source code might look like this: `string EUFormat(); string USFormat();` and its protocol like this: `(EUFormat + USFormat)*`. The names of the interface can be *iDTSFormats* in both cases. The *scTimeLog* subcomponent will provide the logging functionality via the *ILog* (named e.g. *iDTLog*) interface which must be provided by the subcomponent. The added value is that the *scTimeLog* (in our case having a primitive architecture) is able to provide (in its underlying implementation) logging enriched with timestamps formatted according to instructions provided by the *scTimeLog*. Now you must create ties similarly as described above. After it is done, the final picture may look as follows:



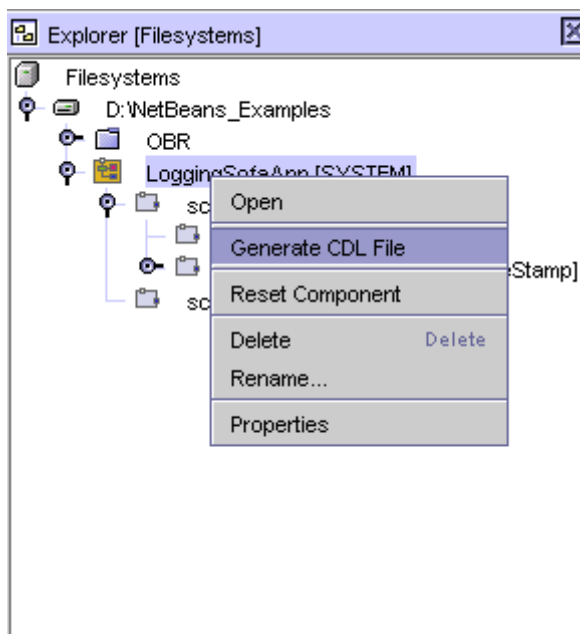
The last action that remains to do is specifying frame protocols of *FDateTimeStamp* and *FTimeLog*. Since the protocols result from the communication on the frames' interfaces, we left it until they were specified. So, fill the following protocols in the *Frame* tabs of the *scDateTimeStamp* resp. *scTimeLog* subcomponents:

```
(?iDTSFormats.EUFormat || ?iDTSFormats.USFormat)*
resp. ?iDTSLog.doLog{!iDTSFormats.EUFormat}*
```

At the very end of the visual component creation process, save

the results into the *LoggingSofaApp.sofa* file using the standard *File->SaveAll* menu items.

Now, you are ready to generate the CDL source file. Click on the node of the highest component and by clicking the right mouse button you bring about the context menu. Choose *Generate CDL File* as indicated in the picture bellow. The CDL file is generated.



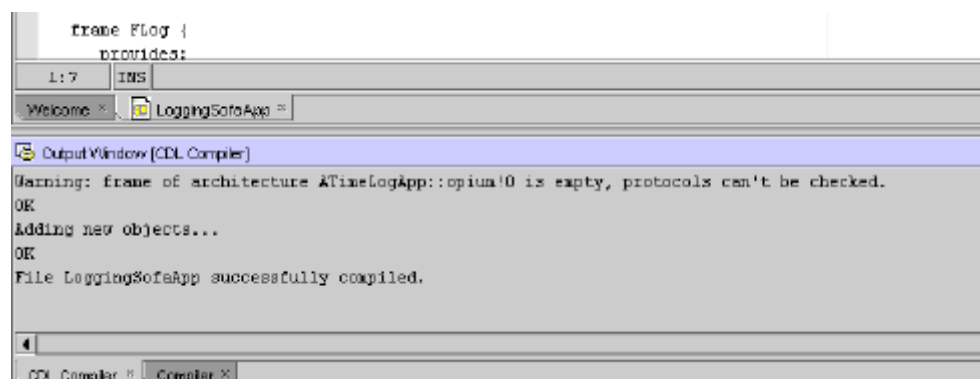
2.4 Compiling CDL Files

Now you have a CDL source file in the editor window of the NetBeans IDE. To be able to compile it, you must be connected to the TIR and the TIR must be set to read-write mode. See [this text](#).

Having focus on the Editor window, go to the *Build* menu of the main pull-down menu of the NetBeans and choose *Compile*. Alternatively you can press the respective hot-key (by default it is F9).

You can watch results of the compilation in the *Output window* which is by default below the *Editor window*. If the compilation fails, the TIR is set to the read-only mode by aborting. So if you want to re-compile, you must set the TIR to read-write mode again (besides correcting the errors, of course). If the compilation passes correctly, the TIR remains in the read-write mode. This is because you are given a choice of either committing the results or aborting them. It must be done explicitly as [described](#) in the next section.

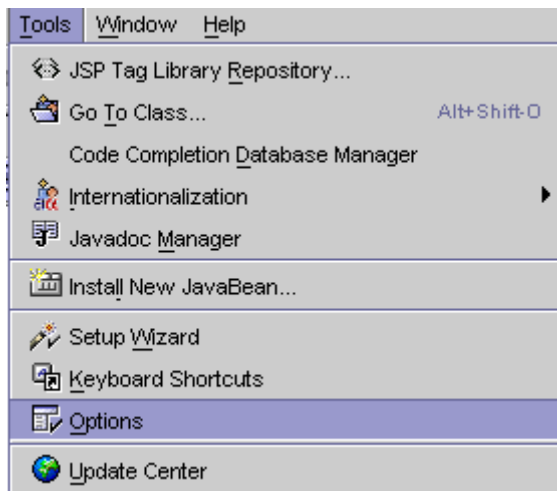
In our case, if you followed the instruction in this text, the compilation should proceed without any error. Ignore warnings about missing protocol in the system frame, it really should not have any.



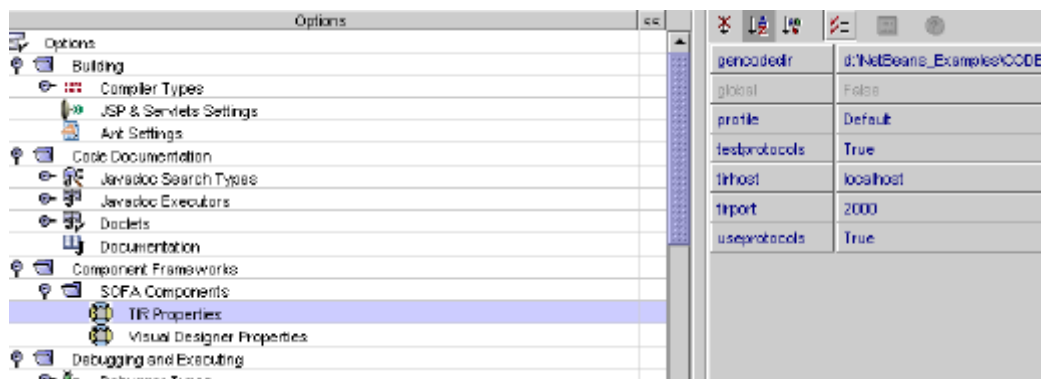
2.5 Using the SOFA TIR Browser

First, the SOFA TIR must be launched⁶. Check whether the TIR properties in the *Tools->Options* menu are set correctly.

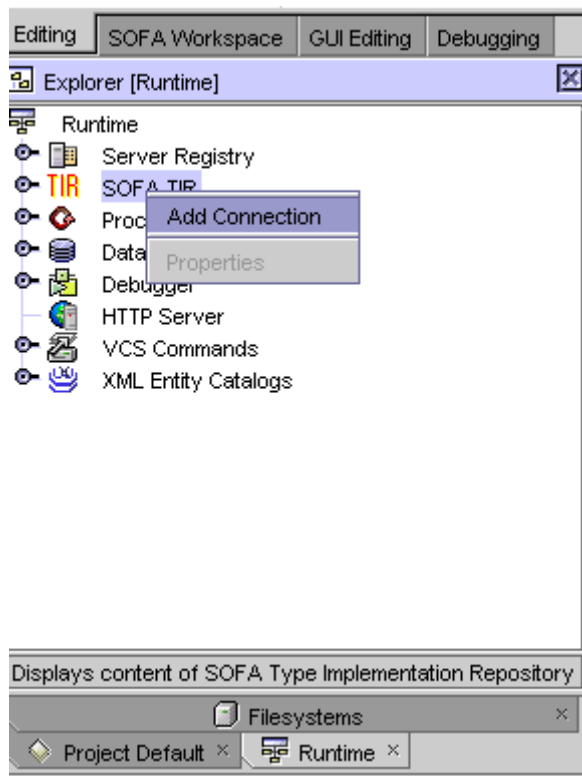
⁶Description how to do it is beyond the scope of this guide



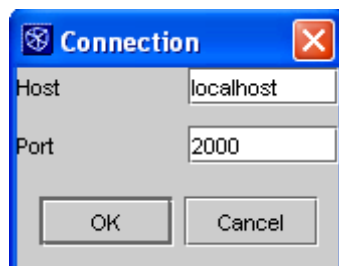
If you follow the LoggingSofaApp example, you can set using and testing protocol to true. A sample setting is shown in the picture bellow.



Switch to the *Editing* workspace and in *Explorer* window to the *Runtime* tab. Select *SOFA TIR* and click the right mouse button to bring about the context menu. From the context menu choose *Add Connection*.



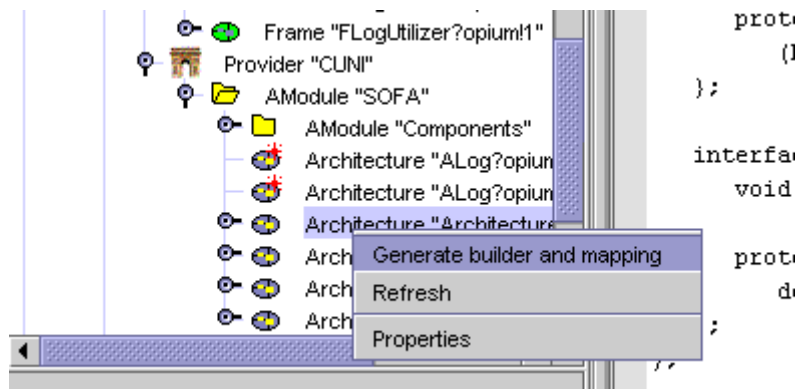
A dialog appears into which you must enter names of the host and port where your RMIRegistry runs.



Under the SOFA TIR node in the Explorer window, you can now unfold sub-nodes and browse the content of the repository. Each node represents an element of the CDL language. Right-clicking on each node, a set context menu with a set of supported actions appears. All nodes support displaying their properties. All nodes of type "container" (i.e. that contains sub-nodes) also support

refreshing their content.

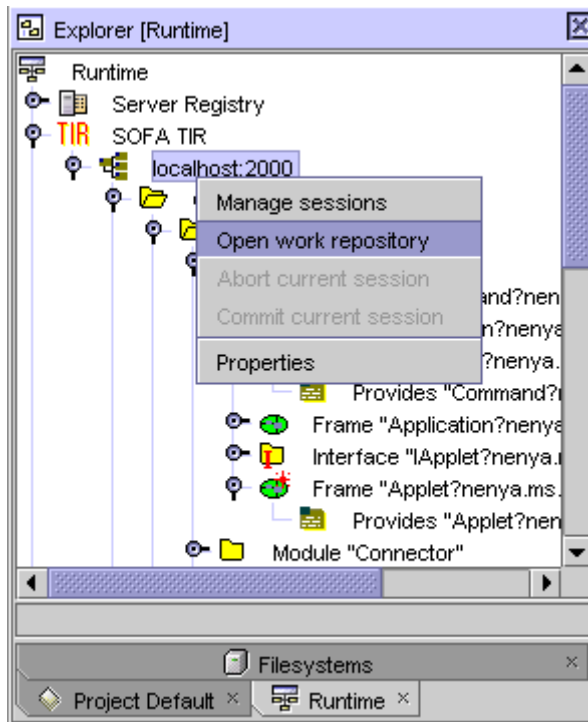
However, the really useful action is the possibility to have the java source code generated from CDL architectures. From historical reasons, architectures are stored in the *Provider* core node. The context menu obtained after right clicking on an architecture node contains *Generate builder and mapping* as an additional item.



Choosing this item, Java mappings of interfaces the architecture contains and a builder which among others contains Java code reflecting ties between the interfaces is generated to the directory that was chosen in options. So if you want to finish the example, choose generation of the code for each of the architectures you have created in this example so far. You are done.

2.5.1 Transactional nature of TIR

An important aspect about the TIR is that it has a transactional manner. By default, the TIR is set to the read-only mode and it cannot be modified. If we want add some additional elements to it, we must set the repository to read-write mode starting a transaction. This can be done by right-clicking the node with the host name and port where the rmiregistry runs. A sample situation is shown in the picture bellow:



When the *Open work repository* action is chosen and proceeds correctly, the icon of the node changes to a sheet with pen. Now we can add additional elements into the repository, which is usually done by compiling a CDL source file. Using the same context menu, we can set the TIR to the read-only mode stopping the transaction by choosing either *Abort current session* or *Commit current session* from the context menu of the host-port node of an set the TIR to the read-write mode.

Chapter 3

TROUBLESHOOTING

3.1 F.A.Q.

3.1.1 Security policy not set

Q: I want to connect to a TIR (using *TIR* -> *Add connection* in the Runtime tab of the explorer window) and the `java.lang.SecurityException` at `org.netbeans.core.execution.TopSecurityManager.checkConnectImpl(...)` appears.

A: It is caused by the fact that you do not have set the security policy for your JVM. The easiest remedy is creating a file `java.policy` in your home directory that grants all permissions ("grant permission java.security.AllPermission"). However, this approach is not recommended due to security holes. For more specific configuration information, see documentation to RMI available within J2SDK.

3.1.2 Profile locked

Q: I want to open TIR for writing but the "Profile locked" exception occurs.

A: This exception may occur when a client crashed or finished

without aborting or committing a transaction in which case the profile remains locked. You can abort the old session using "Manage session" dialog:

1. Go to the Explorer window and select the Runtime tab.
2. Right-click the session node and choose "Manage sessions".
3. Select session you want to kill and click on "Abort connection"