Bachelor Thesis ”VeriNeC Editor GUI”

Damian Vogel damian.vogel@unifr.ch

September 30, 2005

Bachelor of Science in Computer Science

University of Fribourg

Verinec Project

Telecommunications, Networks & Security Research Group

Department of Computer Science

Abstract

This thesis describes the environment and the development of a graphical user interface (GUI) which facilitates configuring computers in the network simulation and configuration program Verinec. This paper gives an overview of the functionality, the implementational choices and decisions, and the programming principles used in the code.

Contents

1 Introduction

1.1 Environment

The Bachelor Thesis "Editor GUI" is part of the Verinec Project of the *telecommunications, networks & security* Research Group TNS of the of the Department of Computer Science at the University of Fribourg.

> The Verinec project aims to simplify network configuration. It is based upon an abstract definition of a network and the nodes in that network expressed in XML. Each node consists of its hardware (network interfaces) and a set of services such as DNS server, firewalls, and so on. The abstract configuration is translated automatically into configuration specific to the actual hard- and software used in the network. The simulator part of Verinec allows to check if the configuration will fullfill the desired behaviour prior to really configure the nodes. [1]

1.2 Assignment

The Verinec project is currently in a state where nodes can be created and interfaces can be added. But if one wishes to change the name of a node or the IP Address of an interface, the XML files have to be edited by hand. My project is to implement a Graphical User Interface (GUI), which facilitates configuration of the nodes and interfaces, similar to a configuration panel in an operating system.

The assignment wording is as follows:

> The VeriNeC GUI is currently capable of displaying a network and the simulator output. One can also create nodes and connect them to a network.
>
> We plan to have editor interfaces for each service of the node to allow the user to edit the node configuration using the GUI. There will be a module for interfaces, DNS Server, Firewall and so on.
>
> This project is to write editor modules for two or three services. You will begin with a simple service and then choose a more complex one.
>
> A simplified example is the
> verinec.networkanalyser.core.DefaultComponentEditor [2]

The technologies to use are

- Java: Swing, JDOM
- Network Technology (i.e. DNS, Firewall, ...)

1.3 Personal Motivation

I have applied for this project mainly because of three reasons. First this project is about concrete programming and implementation. There are no big theoretical researches to be done, and I have to write code which will result in a concrete executable application.

Second, I like GUI specific issues, or more general human – computer interaction issues. I think it is very important to reflect about how we interact with computers and software, and in which way the information is presented by a piece of software. How can a program be intuitive for an user? What behavior is he expecting, and how will he react? These questions I asked myself when implementing the Editor GUI.

And third but not least, the technologies I used within this project are technologies with a certain perspective for future use. Java, or object-oriented programming languages in general, are basics for a computer scientist today. The same applies to XML and XML-based technologies such as JDOM or XSLT (XSLT I intended to use but abandoned due to a lack of time).

2 Functionality

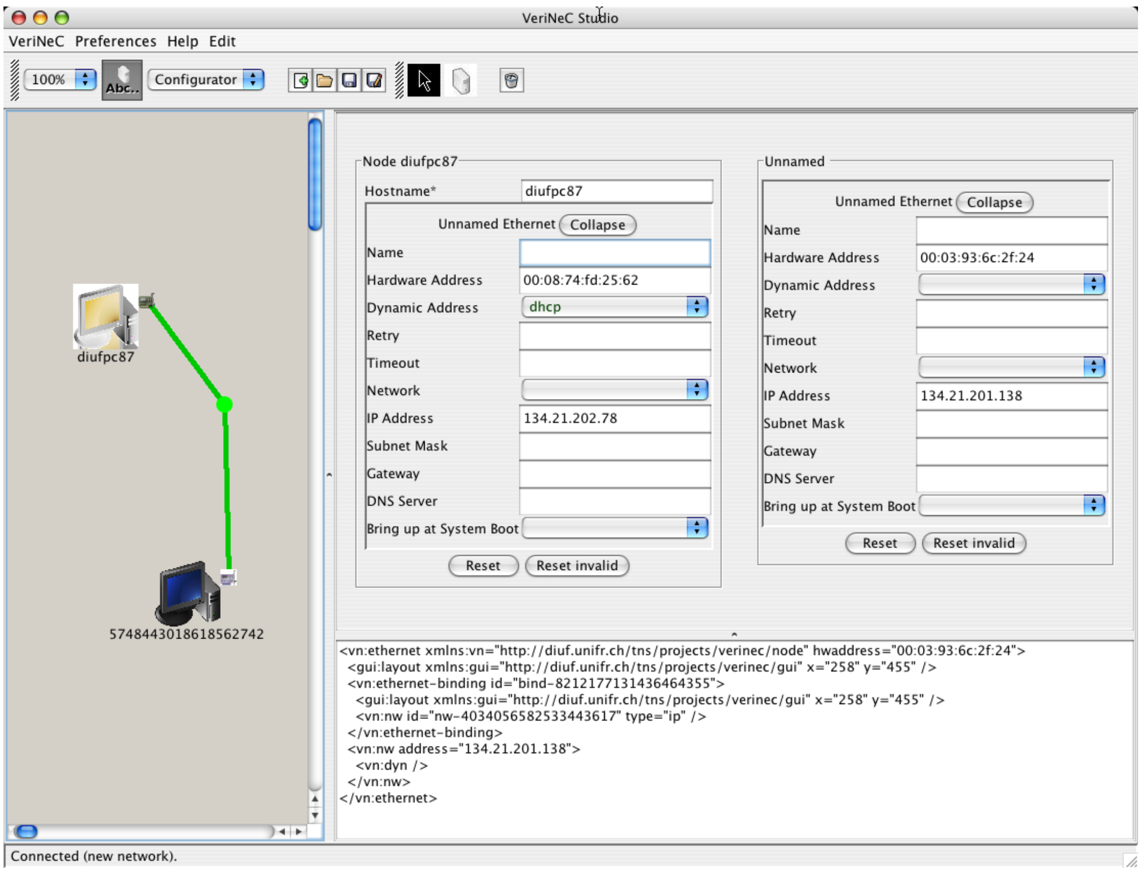
The functionality of the Editor GUI is to accept information from the user and store it in an XML structure. The information that has to be treated is very similar to the information when configuring a network interface in the configuration panel. The user can configure a computer (in this project mostly called *node*) with name, IP Address, Gateway Server, and so on.

A node can have multiple interfaces of different types, an interface can have multiple connections or, to use the usual name within this project, multiple *bindings* (except for the ethernet interface which only has one binding).

The following functionality had to be achieved and principles had to be respected:

- It has to be clear which interface or binding the user is working on
- The input has to be verified upon formal correctness before being saved to the XML structure
- A hint can be displayed to help the user find the correct form
- Incorrect inputs can be reset to the last correct value in case the user is completely lost
- Not-needed information (i.e. a binding or an interface) can be hidden and revealed on demand
- Multiple nodes or interfaces can be edited at the same time, facilitating comparison of input values
- Reduce to the max: no unnecessary mouse clicks or dialogue boxes should yield efficient workflow

Figure 1: Screen shot of the NodeEditor



Node diufpc87

Hostname*	diufpc87

Unnamed Ethernet Collapse

Name	
Hardware Address	00:08:74:fd:25:62
Dynamic Address	dhcp
Retry	
Timeout	
Network	
IP Address	134.21.202.78
Subnet Mask	
Gateway	
DNS Server	
Bring up at System Boot	

Reset Reset invalid

Unnamed

Unnamed Ethernet Collapse

Name	
Hardware Address	00:03:93:6c:2f:24
Dynamic Address	
Retry	
Timeout	
Network	
IP Address	134.21.201.138
Subnet Mask	
Gateway	
DNS Server	
Bring up at System Boot	

Reset Reset invalid

```
<vn:ethernet xmlns:vn="http://diuf.unifr.ch/tns/projects/verinec/node" hwaddress="00:03:93:6c:2f:24">
  <gui:layout xmlns:gui="http://diuf.unifr.ch/tns/projects/verinec/gui" x="258" y="455" />
  <vn:ethernet-binding id="bind-8212177131436464355">
    <gui:layout xmlns:gui="http://diuf.unifr.ch/tns/projects/verinec/gui" x="258" y="455" />
    <vn:nw id="nw-4034056582533443617" type="ip" />
  </vn:ethernet-binding>
  <vn:nw address="134.21.201.138">
    <vn:dyn />
  </vn:nw>
</vn:ethernet>
```

Connected (new network).

3 Implementation

The code can be structured in two main parts: object-oriented classes and functional (static) classes. Actually the code of the static classes could be integrated within an other class, but to me it seems more tidy to group the code belonging to a common theme in a special class.

My personal goal is to create elegant code. I cannot say I always achieved this, but it might be that I sometimes preferred an in my eyes ”proper” solution over a code with better performance.

3.1 EditorComponent Classes

The object-oriented classes rely on one class: the EditorComponent interface class. This Interface specifies that every panel, text field or combo box used in the EditorGUI will implement the functions *update()*, *save()*, *discard()*, *discardRed()* and *add(Component)*. By unifying the functionality of the different components, I can care less about whether I work with a panel or a single text field.

This is an implementation of the structural design pattern ”Composite”. As a short reminder what the Composite Pattern is about:

> Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. [3]

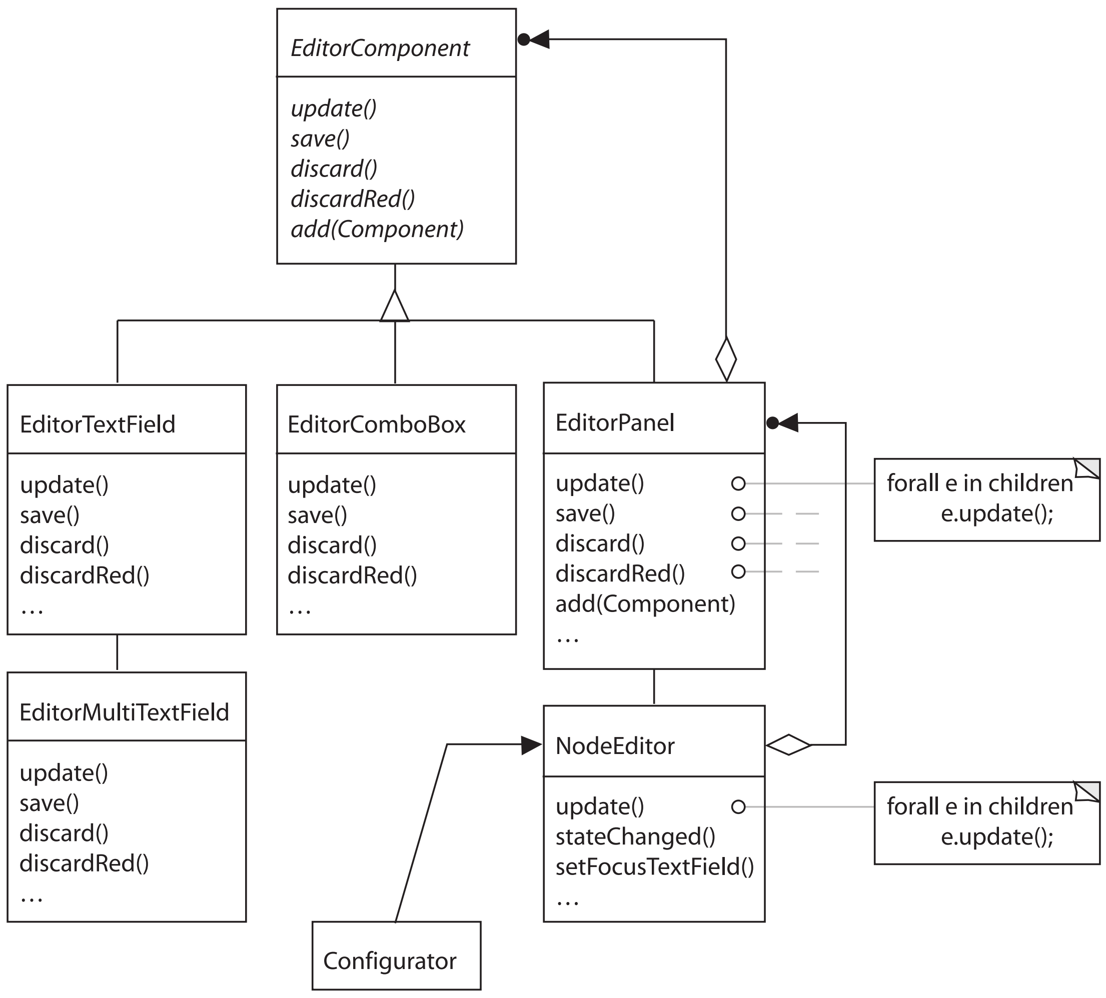
The classes shown in Figure 2 correspond with the following prototypes:

EditorComponent	Component
EditorPanel	Composite
NodeEditor	Composite
EditorTextField	Leaf
EditorComboBox	Leaf
Configurator	Client

3.2 EditorPanel

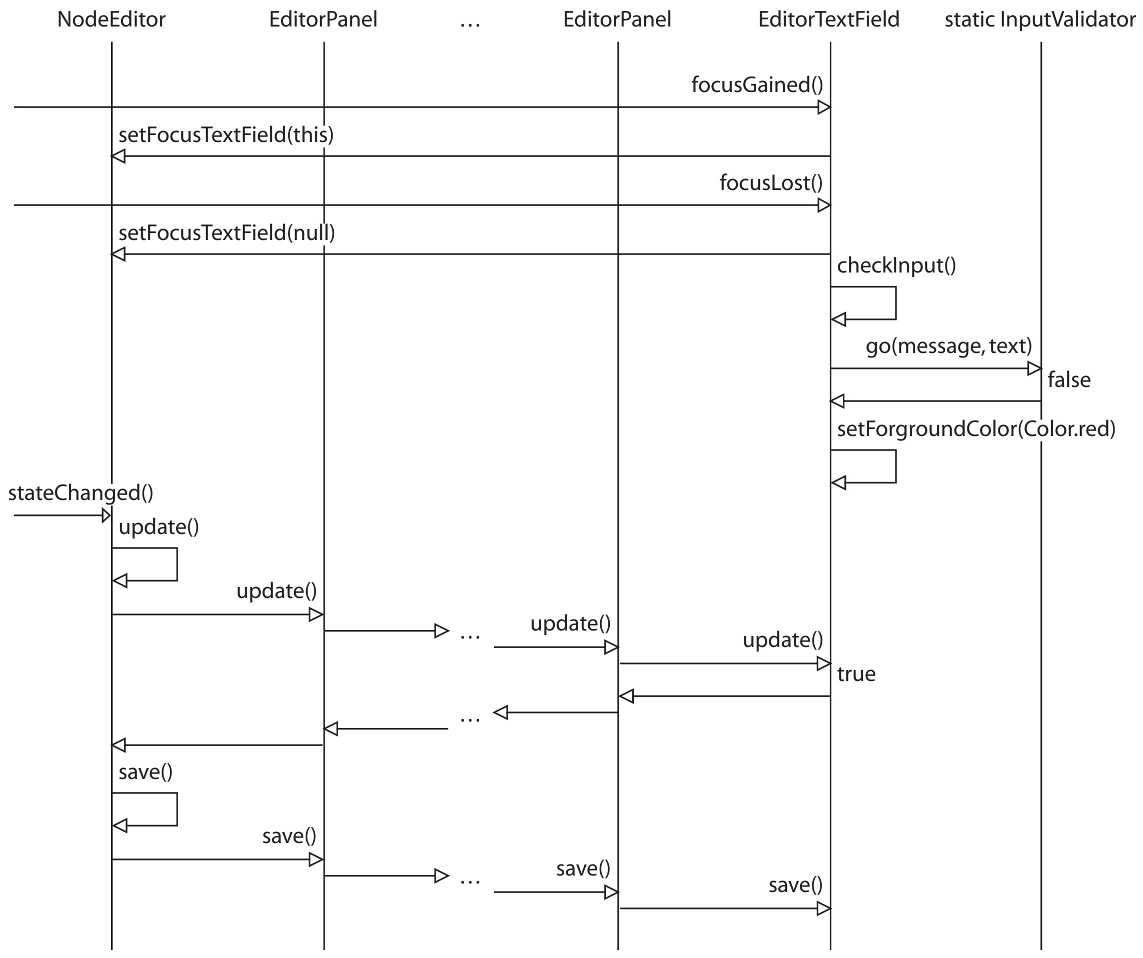
The EditorPanel extends JPanel and implements the EditorComponent; its used for all panels in my GUI. This is necessary to make the Composite Pattern (see Figure 2) work.

Figure 2: UML diagram of the EditorComponent interface and its implementing classes.



The ***update()* function** is called in the *stateChanged()* function which is triggered when the user changes the selection; it calls the *update()* function within each child EditorComponent. The iteration of *update()* function is stopped as soon as a modified input is encountered, and the boolean value *true* is returned. The last parent panel, which is always the NodeEditor [3.6], calls the *save()* function of all its children if *true* is return. The *save()* function proceeds analogous to the *update()* function. Figure 3 illustrates a typical call sequence.

Figure 3: Call sequence from NodeEditor to EditorTextField.



Besides the functions necessary to implement the Composite Pattern, the EditorPanel holds the following functions:

- Functions to add ”Reset” buttons
- Action Listener to catch when a button is hit
- Functions to add a ”Collapse” button
- Functions to determine a ”collapsible” panel and change the state of it (i.e. collapse or show)

3.3 EditorTextField

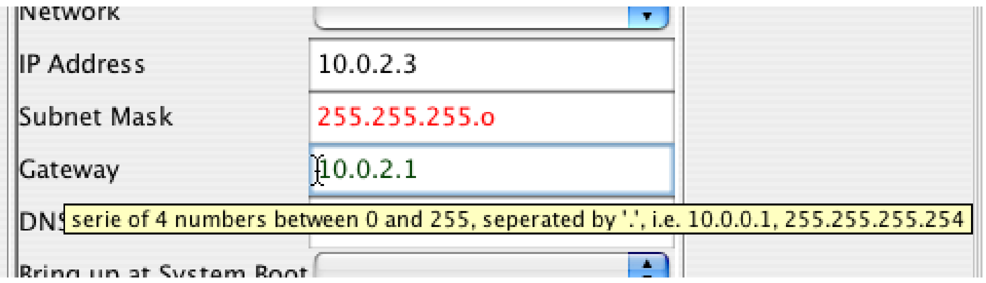
The EditorTextField extends JTextField and implements the EditorComponent interface. The EditorTextField is a "Leaf" class in the Composite design pattern (Figure 2).

On creation, it receives an InputValidator [3.8] class field that serves as InputValidator pattern, and depending on the pattern a ToolTip string is attributed (see Figure 4). It receives a Label string which is stored as an instance field and will be displayed next to the field.

Now once the field **gets focus**, it registers itself in the NodeEditor class as "the field having focus" by calling the *setFocusTextField()* function. This is necessary because the FocusLostEvent is not triggered if the component gets unselected in the DrawPanel, and a modified input might be lost.

When a field has focus, there can be two events: either an **ActionPerformedEvent**, or an FocusLostEvent. The ActionPerformedEvent occurs when the user hits the *Enter* or *Return* key; the effect is that the input is verified and the field changes color according to the verification result; the focus remains with the current field. (This behaviour could be changed easily by activating a code line.) The **color code** for the field is illustrated in Figure 4: An unmodified field is black. If the value is changed and happens to be the same value as before the modification, it stays black. If a value is entered which fails the verification, the field turns red. If the value is correct but differs from the previous value, it turns green, and the value is written to the underlying XML.

Figure 4: EditorTextFields change color after modification.



Network

IP Address 10.0.2.3

Subnet Mask 255.255.255.o

Gateway 10.0.2.1

DNS serie of 4 numbers between 0 and 255, seperated by '.', i.e. 10.0.0.1, 255.255.255.254

Bring up at System Boot

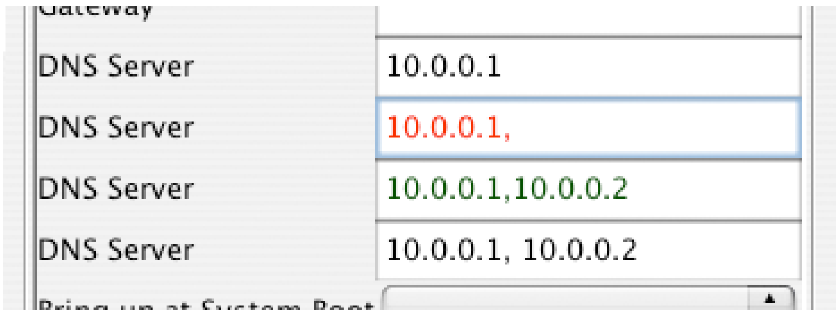
The **FocusLostEvent** occurs if the *Tab* key is hit or the focus is transfered using the mouse. The Effect is the same as with the ActionPerformedEvent; the data is validated and according to the result the field changes its color. Additionally, it unregisters in the NodeEditor class.

When **writing the XML** data, there is an issue worth noting: If the input data is empty, two different situations can occur: either the XML attribute is necessary, then the value will just be reset to the previously saved (and therefore correct) value. These text fields have an asterisk next to their name. Or the XML attribute is optional, which is far more often the case, then it will be removed.

3.4 EditorMultiTextField

The EditorMultiTextField extends the EditorTextField and inherits most functionality from it directly such as the color feedback of the InputValidator or the XML treating of not existing nodes. Also the functions *update()*, *save()*, *reset()* and *resetRed()* are executed in the superclass.

Figure 5: Different stages of an EditorMultiTextField



DNS Server	10.0.0.1
DNS Server	10.0.0.1,
DNS Server	10.0.0.1,10.0.0.2
DNS Server	10.0.0.1, 10.0.0.2

1. EditorMultiTextField with only one value
2. After adding a separator, the string is not valid any more
3. Completion with a second value
4. The pattern accepts both variants, with or without space after the comma

The EditorMultiTextField adds an important behaviour though: it can accept a serie of values with a separator in between, as you can see in Figure 5. The EditorMultiTextField receives a regular expression string for separation, and a string for concatenation on creation. This allows the EditorMultiTextField to accept lists of values which it decodes and stores into separate XML elements. The pattern strings are defined as constants in the InputValidator [3.8] class.

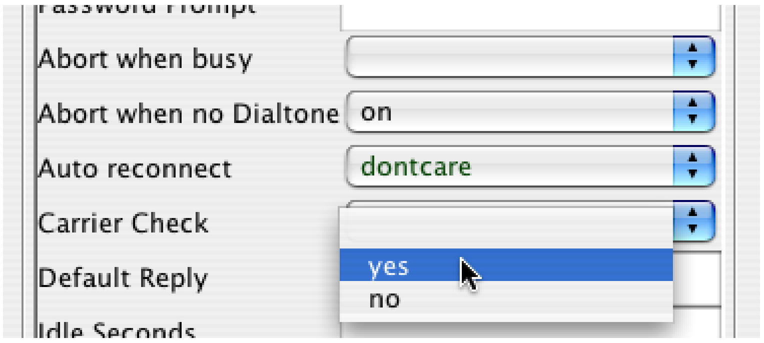
The functionality is achieved by using an intermediate function to read and write the XML data. In the EditorTextField, the intermediate functions simply translate the command and return the result. In this class, the functions contain an algorithm to read or create several elements and set an attribute.

3.5 EditorComboBox

The EditorComboBox extends JComboBox and implements the EditorComponent interface. It works very similar to the EditorTextField, except that there is no need for validation of the input data. It also changes color upon modification (Figure 6) and creates or removes XML attributes if necessary.

The array holding the strings to choose from is a constant in the InputValidator class and is passed as an argument on creation.

Figure 6: EditorComboBox changes color after modification.


Password Prompt
Abort when busy
Abort when no Dialtone: on
Auto reconnect: dontcare
Carrier Check
Default Reply
Idle Seconds

3.6 NodeEditor and DisplayComponent

The NodeEditor class extends the EditorPanel class. It is instantiated in the Configurator class' *load()* function (see Appendix C). The DisplayComponent class is a purely static class which provides functionality for the NodeEditor class. Actually the code could be merged in one class, but in order to improve legibility and comprehensibility it has been separated.

The NodeEditor receives the **ChangeEvents** from the Configurator class when a node, interface or binding is selected or unselected in the Draw Panel. It maintains a Hash Map to keep track of the currently selected panels.

The inherited *update()* function to iterate the child components is overridden and replaced by a slightly modified function which calls the *save()* function when receiving true from one of its children instead of returning it (see Figure 3).

For the description of the *setFocusTextField()* function see section 3.3 EditorTextField.

Now to speak of the **DisplayComponent class**: this is where the detail work is done and all the EditorPanels, EditorTextFields or ComboBoxes and Buttons get instantiated. The functions are ordered hierarchically, actually a pure functional top-down approach. The entry point is the *DisplayComponent()* function which is called from the Editor Node on a change Event. This function creates a first panel including a frame and then checks the XML for certain criteria via static functions of the ElementType [3.7] class. After detecting what kind of component it is treating, it calls the appropriate function. This next function adds certain input fields and again checks what other components exist in the XML it has received, and calls other follow-up functions again.

To **illustrate the functionality** of the DisplayComponent class, let me show you some simplified code excerpts: In Figure 7 is the code of the *DisplayComponent()* function. It receives an XML element as argument, and then analyzes it. Lets say the XML is a node thus it calls the *displayNode()* function on *line 2* with the element as argument again.

Figure 7: Simplified code of the DisplayComponent class: displayComponent() and displayNode()

```
 1 displayComponent(baseElement){
 2     if (isNode(baseElement)) displayNode(baseElement);
 3     else if (isWlan(baseElement)) displayWlan(baseElement);
 4     else if (...)
 5 }
 6 displayNode(baseElement) {
 7     EditorTextField hostnameField = new EditorTextField(...);
 8     fieldPanel.add(hostnameField);
 9     labelPanel.add(hostnameField.getJLabel());
10     List wlanList = baseElement.getChildren(Wlan);
11     Iterator wlanListIter = wlanList.iterator();
12     while (wlanListIter.hasNext()) {
13         displayWlan((Element) wlanListIter.next());
14    }
15 }
```

The *displayNode()* function then creates an EditorTextField *(line 7)* which will hold the hostname, adds the field *(line 8)* and its label *(line 8)* to the respective panels, and looks for further components in the XML *(line 10)*. As long as the list has elements *(line 12)*, it will call the *displayWlan()* function with the corresponding child element as argument *(line 13)*.

3.7 Static Class: ElementType

In this class, all the functionality for the XML treatment is grouped. In a first part the strings corresponding to the names of XML elements and attributes are defined as constants (final static). In a second part, there are functions to check whether a given element or string is of a certain kind.

The motivation to group this together in a separate class as constants is that if once in the future the name of an element or an attribute is changed (which actually shouldn't but nevertheless is perfectly possible) then the only place you have to modify the code is in this file. All other classes doing XML treatment rely on this class.

3.8 Static Class: InputValidator

This class contains the functionality needed to check the input of the Editor-TextFields [3.3] and to define the values for the EditorComboBox [3.5].

If you look at the code, it might at first be a little confusing why some constants are defined in this class, and others are defined in the ElementType class. But if you look a little closer you can soon see that this class has nothing to do with the XML structure but only with the datatypes of the attributes.

For most of the datatypes, such as IPv4 address or MAC address, there is a regular expression given in the XML Schema
$PROJECT-ROOT$/java/res/base_types.xsd. But for some datatypes, native XML datatypes could be used. These types and their definition can be found on the web under [4].

The regular expression (pattern) for verification of the *string* type and the *NMTOKEN* type has been taken from there. For the pattern of *unsignedLong* and *integer* I had to improvise a little. According to the W3C:

> [Definition:] **integer** is derived from decimal by fixing the value of fractionDigits to be 0. This results in the standard mathematical

concept of the integer numbers. The value space of integer is the infinite set {...,-2,-1,0,1,2,...}.

integer has a **lexical representation** consisting of a finite-length sequence of decimal digits (#x30-#x39) with an optional leading sign. If the sign is omitted, "+" is assumed. For example: -1, 0, 12678967543233, +100000. [4]

Let me put it in my own words: it is an arbitrary integer, optionally preceded by '-' or '+'. The regular expression to this was not that hard:

either:	0
or:	a number not beginning with 0
or:	'+' or '-' followed by a number not beginning with 0

"$0|[1-9][0-9]*|([+]|[\backslash -])[1-9][0-9]*$"

The verification of the unsigned long was more tricky: it has a scope of 0 to $(2^{64}-1)$, which is bigger than any number Java can handle: in Java, only "Long" exists, scope -2^{63} to $(2^{63}-1)$, but no "unsignedLong". Therefore it could not be parsed into a Long as a verification. Instead I chose to approximate the value of $(2^{64}-1) = 18'446'744'073'709'551'615$.

either:	0
or:	a 1-digit to 19-digit number not beginning with digit 0
or:	a 20-digit number beginning with digit 1 and followed by a digit which is smaller than 8

"$0|[1-9][0-9]\{0,18\}|1[0-7][0-9]\{18\}$"

This pattern allows me to express numbers from 0 to
17'999'999'999'999'999'999
instead of parsing it as Long and hitting the ceiling at
9'223'372'036'854'775'807

4 Conclusion

The following is a list of suggestions, in what way the GUI, the Analyser or Verinec as whole could be changed.

4.1 Verinec General Issues

”Show Parent”

When beginning this project, I wanted the the parent element being accessible from within the configuration panel, i.e when selecting an interface there is a button ”Show Parent”. After I saw a little how the Analyser class worked I soon had to abandon this idea: When the selection is changed, the NwComponent that is transferred has no reference to its parent Component. Though this functionality entails quite a lot of work, it would definitely be an item on my wish list.

4.2 Future Editor GUI Development

Add ...

When using Verinec on my iBook, I had some troubles to add new interfaces to a node. Apple's PowerBooks or iBooks do not have any right mouse button next to the track pad, to get a context menu the user can press the CTRL-key as a modifier key. This doesn't work with Java applications if they do not intentionally feature this behaviour. Therefore it was planned to add a button to the NodeEditor [3.6] which gives the user the same functionality as a right mouse click does.

Maybe it could be implemented by adding a button which on click presents the context menu as if the user pressed the right mouse button on the corresponding component.

NodeEditorPanel position

Although experimenting a lot with *setSize()*, *setPosition()* and different Layout Managers I was unable to prevent the NodeEditorPanel from centering vertically. This doesn't cripple its functionality, but it would be nicer if it would stay aligned on top all the time.

XSLT Validator And ToolTip

A thing which I would have liked quite a lot is the implementation of an XSL transformation sheet which gets the regular expression pattern from the nodes.xsd into a Java file. The same could be done with the ToolTip which could also be stored in the XML schema file (additionally to the xs:documentation tag).

Configuring Services

And not to leave the assignment incomplete I will comment on the configuration of services. I agreed with my tutor and my professor that creating configuration panels for different services would exceed my bachelor thesis. But sure it would be handy to configure a firewall, DNS server and so on directly from within the Configurator panel.

4.3 Personal Closing Words

Personally I have liked the work on the project a lot. In the beginning it was hard because the project was difficult to set up correctly. But as soon as I went to get help from my tutor it worked, it was a personal problem that I didn't seek help early enough.

Once I got started with programming, there was a long dry spell until I was able to see the grapes of my labour, for example the Composite design pattern in the EditorComponent classes [3.1] had to be implemented completely before it could execute.

I want to thank my tutor David Buchmann for supporting me with ideas and help when I was stuck with my code and struck with blindness for my errors. I think I have learned a lot about programming with object-oriented programming languages and the respective concepts.

A Bibliography

References

[1] Telecommunications, Networks & Security Research Group, 2005
Verinec Translation Module (Verified Network Configuration - Working Paper), state September 18, 2005
by David Buchmann and Dominik Jungo
http://diuf.unifr.ch/tns/projects/verinec/verinec_report.pdf

[2] Telecommunications, Networks & Security Research Group, 2005
Editor GUI Bachelor Thesis Assignment
by David Buchmann
http://diuf.unifr.ch/tns/

[3] Addison Wesley Professional Computing Series, 1994
Design Patterns: Elements of Reusable Object-Oriented Software
by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

[4] W3 Consortium, 2001
XML Schema Part 2: Datatypes
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

B Licensing

C Code modification in the Configurator class

The following lines of code are suggested to be added to the *public void load(Analyser parent)* function in the Configurator class in order to load the Editor GUI:

```
public void load(Analyser parent) throws VerinecException {
    (...)

    Component xmlData = new DefaultComponentEditor(analyser);
    JScrollPane xmlScrollPanel = new JScrollPane(xmlData);

    NodeEditor nodeEditor = new NodeEditor(analyser);
    JScrollPane nodeEditorScrollPanel = new JScrollPane(nodeEditor);

    analyser.setTopComponent(nodeEditorScrollPanel);
    analyser.setBottomComponent(xmlScrollPanel);

    (...)
}
```