Bachelorarbeit

Mike Brasch

Entwicklung eines modularen Prüfsystems für komplexe Sensorik

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Mike Brasch

Entwicklung eines modularen Prüfsystems für komplexe Sensorik

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer nat. Stefan Pareigis
Zweitgutachterin: Prof. Dr. Zhen Ru Dai

Abgegeben am 2. Mai 2012

Mike Brasch

Thema der Bachelorarbeit

Entwicklung eines modularen Prüfsystems für komplexe Sensorik

Stichworte

Rich Client Platform, Netbeans Platform, Skript, BeanShell, Wizard, Sensor

Kurzzusammenfassung

In der vorliegenden Arbeit wurde eine modulare Software-Umgebung zum automatisierten Testen einfacher und komplexer Geräte entwickelt. Diese erlaubt es, mit Hilfe skriptgesteuerte Wizards dynamische Tests durchzuführen. Weiterhin wurde anhand einer beispielhaften Implementierung ein automatischer sowie ein manueller Testdurchlauf eines Sensors durchgeführt.

Mike Brasch

Title of the paper

Development of a modular test system for complex sensors

Keywords

Rich Client Platform, Netbeans Platform, script, BeanShell, wizard, sensor

Abstract

In this paper, a modular test system for automated testing of simple and complex devices is developed. With help from script controlled wizards it is possible to execute dynamic tests. Furthermore a first implementation allowed the execution of an automated and also of a manual test run of a sensor.

Inhaltsverzeichnis

Abbildungsverzeichnis **5**

1 Einleitung **6**

2 Anforderungen **7**

3 Designentscheidungen **9**

3.1 Rich Client Platforms 9

NetBeans Platform 9

3.2 Skriptanbindung 10

Beanshell 10

Erweiterte Sprachfeatures gegenüber Java 10

Einschränkungen und Nachteile der BeanShell 12

4 Das modulare Testsystem Sopas TT **13**

4.1 Main-Modul 13

4.2 ScriptableWizard 14

4.3 BeanShell 17

4.4 BitIO 17

4.5 ProtocolWriter 18

5 Testablauf **20**

5.1 Wizard-unterstützter automatischer Test 20

5.2 Wizard-unterstützter manueller Test 24

6 Fazit und Ausblick **26**

Literaturverzeichnis **27**

Anhang **28**

1 BeanShell-Skript: Kabeltester 28

Abbildungsverzeichnis

Abbildung 1: Beispiele für Swing-Elemente in einem Panel..... 15
Abbildung 2: Remote-IO-Module (BitIO)..... 20
Abbildung 3: Kabeltester-Box..... 21
Abbildung 4: Wizard – Startseite mit Testbeschreibung..... 21
Abbildung 5: Wizard – Angaben zum Test..... 22
Abbildung 6: Wizard – Anschlußanweisungen..... 22
Abbildung 7: Wizard – Testfortschritt..... 23
Abbildung 8: Wizard – Schlußseite, abschliessende Bemerkungen..... 23
Abbildung 9: Platine mit Taster und LED..... 24
Abbildung 10: Wizard – Handlungsanweisungen..... 24
Abbildung 11: Wizard – Abfrage..... 25

1 Einleitung

Diese Bachelor-Arbeit wird bei der SICK AG in Hamburg entstehen. SICK ist ein Anbieter von Sensorensystemen, mit Hauptsitz in Waldkirch, Baden-Württemberg.

Ziel ist die Entwicklung einer Softwareumgebung zum automatisierten Testen einfacher bis komplexer Sensoren, die über Schnittstellen, wie z.B. Ethernet oder USB, kommunizieren. Die Testmöglichkeiten richten sich hierbei nach dem, was die Sensoren über ihre APIs zur Verfügung stellen. Dadurch kann nicht getestet werden, ob die physikalische Sensorik korrekt arbeitet. Es wird nur das externe Verhalten des Sensoren geprüft.

Notwendig ist ist diese Software, um das Testen der Sensoren zu vereinfachen und durch eingewiesene Personen zu ermöglichen.

Im Folgenden werden die Anforderungen an die Software erörtert, um darauf aufbauend Designentscheidungen für die Entwicklung treffen zu können. Maßgebend hierfür ist die Auswahl einer Rich Client Platform und einer Skriptsprache. Das daraufhin entwickelte modulare Testsystem wird ausführlich dokumentiert. Anhand dieser Beispielimplementierung werden unterschiedliche Tests entwickelt und deren Ausführung automatisiert. Abschliessend wird ein Fazit gezogen und ein Ausblick auf weitere Entwicklungsschritte geworfen.

2 Anforderungen

Die Anforderungen sind zunächst einmal nicht sehr streng definiert, so dass es kein Pflichtenheft im eigentlichen Sinne gibt. Vielmehr soll zunächst untersucht werden, was sich testen lassen würde und was praktisch durchführbar wäre. Die Software soll modular sein und mehrere Geräte gleichzeitig bedienen können. Sie soll sowohl die zu testenden Sensoren, als auch Geräte, die auf die Prüflinge einwirken, (z.B. eine Rüttelplatte, Klimakammer etc.) bedienen können.

Gegenstand der Tests sind im wesentlichen einzelne Sensoren, auf die unterschiedlich eingewirkt wird. Man könnte z.B. das Verhalten eines Sensors beim Überschreiten oder Unterschreiten eines Grenzwertes beobachten. Dann, sobald das Gerät Aussetzer zeigt, wieder in den zulässigen Bereich zurückkehren und beobachten, ob der Sensor wieder in den Normalbetrieb übergeht und wie viel Zeit es in Anspruch nimmt. Grenzwerte können u.a. sein:

- Überschreitung der zulässigen maximalen Arbeitstemperatur
- Überschreitung der zulässigen maximalen Erschütterung (Dauer, Amplitude)
- Unterschreitung des zulässigen Spannungsbereichs

Getestet wird hier der Status von Geräten. Mit Hilfe der vom jeweiligen Gerät angebotenen API kann der Test Werte setzen/abfragen, Methoden aufrufen, Nachrichten abonnieren. Die vom Gerät verwendete API legt den möglichen Testumfang fest. Was von der Geräte-API nicht vorgesehen ist, kann nicht getestet werden.

Funktionstests können nur sehr eingeschränkt durchgeführt werden. Bitfehler können von den Protokollschichten erkannt werden. Wenn aber ein Sensor nicht auf Anfragen reagiert, kann das Testsystem nicht erkennen, ob die Verbindung defekt ist oder der Sensor nicht korrekt arbeitet. Darüber hinaus kann die physikalische Messung des Sensors nicht überprüft werden.

Wünschenswerte Eigentschaften der endgültigen Anwendung sind:

- Der Testablauf soll über ein Skript gesteuert werden können. Die eigentliche Testlogik steckt im Skript, die Anwendung dient nur als Umgebung, die neben der Anzeige auch die gerätespezifischen Dienste bereit stellt.
- Es soll eine Benutzerführung mit Hilfe von einem Wizard möglich sein, welcher vom Skript erzeugt und gesteuert wird.

- Die Anwendung soll modular und leicht erweiterbar sein. Die Unterstützung neuer Geräte und Geräteeigenschaften soll über Module (Plugins) bereitgestellt werden.
- Es soll die Kommunikation über unterschiedliche Schnittstellen möglich sein, z.B. Ethernet, CAN, USB oder Serielle.

Bezogen auf den Umgang mit der Software sind im wesentlichen drei Benutzerrollen vorstellbar:

- **Modul- und Anwendungsentwickler:** Dieser Entwickler betreibt im Wesentlichen die Anwendungspflege. Das beinhaltet auch das Erweitern um Module zu neuen Geräten, oder neue Darstellungsmodule. In den Fällen, wo ein Skripten der Tests uneffizient ist, wie z.B. bei sehr hohem Datenaufkommen, würde er auch entsprechende Tests in Modulform entwickeln.
- **Testentwickler:** Der Testentwickler schreibt Tests in Form von Skripten. Die Vorteile von geskripteten Tests sind u.a., dass der Testschreiber nicht den Quellcode der Anwendung benötigt, sondern nur die Schnittstellenbeschreibung der Module. Außerdem benötigt er nicht unbedingt Kenntnisse über die Anwendung selbst.
- **Tester**: Bei dem Tester kann es sich zum einen um einen Entwickler handeln, der einzelne Aspekte seiner Änderungen am Sensor-Code während der Entwicklungsphase selbst testen möchte. Es kann aber auch eine eingewiesene Person sein, die beispielsweise in der Produktion Abschlusstests durchführen soll.

3 Designentscheidungen

Für die Realisierung der Test-Anwendung lautet die Vorgabe, auf gängige Techniken zu setzen. Auf eine vollständige Evaluation der Alternativen wird hier allerdings verzichtet, da dies, aufgrund des Umfanges, den Rahmen dieser Arbeit sprengen würde. In den folgenden Abschnitten werden die getroffenen Entscheidungen, bezüglich der Auswahl der Rich Client Platform und der Skriptsprache, begründet.

3.1 Rich Client Platforms

Als Basis für die Anwendung bietet sich eine Rich Client Platform (RCP) an. RCPs sind aufgrund ihrer Modularität leicht zu erweitern und bringen alle wichtigen Grundfunktionen, wie z.B. Fenster-Handling, Plugin-Verwaltung und Update-Management, bereits mit. Darüber hinaus werden diese Komponenten von Dritten weiterentwickelt. Die eigene Entwicklung findet in Form von Plugins bzw. Modulen statt.

Als Nachteil steht dem gegenüber, dass eine nicht unerhebliche Einarbeitungszeit für die Konzepte einer RCP anfällt. Aber bereits für mittelgroße und längerfristige Projekte egalisiert sich der Nachteil durch die auf die eigenen Plugins fokussierte Entwicklung.

Es gibt mehrere RPCs, wovon NetBeans Platform und Eclipse RCP wohl mit Abstand am weitesten verbreitet sind, was unter anderem an ihrem Alter und Reifegrad liegen dürfte. NetBeans IDE und Eclipse IDE benutzen ihre jeweiligen RPCs als Basis. Die Wahl für die eine oder die andere Platform dürfte im Wesentlichen an den äußeren Rahmenbedingungen, liegen, z.B.: Was wird schon eingesetzt? Gibt es die gewünschten Plugins? Ein Stück weit dürfte es auch eine Geschmacksfrage sein. Gut sind beide Plattformen.

Für C++ gibt es kaum ausgereifte RCPs und da die Software auch unter Linux lauffähig sein soll, kommen RCPs auf Basis von z.B. .Net nicht in Frage. Somit fällt die Wahl auf eine Java-RCP und dadurch auch auf Java als Sprache.

NetBeans Platform

Die Wahl fällt aufgrund der folgenden Vorteile auf die NetBeans Platform [NBP]:

NetBeans Platform nutzt hauptsächlich Technologien aus den Standard-Java-Frameworks. So wird z.B. Swing zur Darstellung der GUI genutzt. NetBeans IDE bietet mit dem eingebauten GUI-Editor Matisse einen guten WYSIWYG-Editor. Mit dessen Hilfe kann man Swing-Elemente via Drag’n’Drop im Fenster frei anordnen. Guides unterstützen

dabei, dass die Elemente einen korrekten Abstand zueinander bekommen. Da Swing benutzt wird, können ohne Probleme eigene vorhandene Swing-Klassen verwendet werden.

Im Normalfall ist NetBeans IDE für NetBeans Platform die Entwicklungsplattform der Wahl. Sollte es erforderlich sein, könnte die Entwicklung aber auch auf Eclipse oder IntelliJ fortgeführt oder parallel auf einer dieser IDEs entwickelt werden. Bei Eclipse RCP wäre man auf Eclipse IDE festgelegt.

NetBeans Platform fällt unter die CDDL- bzw. (L)GPL-Lizenz, welche sich für die geplante Aufgabe eignen.

3.2 Skriptanbindung

Die Anwendung soll scriptbar sein, damit auch ohne den Quellcode der Anwendung Tests geschrieben werden können. Außerdem muss sich der Testschreiber nicht erst in die RCP-Anwendung einarbeiten. In vielen Fällen wird es reichen, eines der existierenden Skripte zu modifizieren.

Für Java gibt es viele Sprachen, die man als Skriptsprache einbinden kann. Die Wahl fällt dabei letztlich, aufgrund folgender Eigenschaften, auf Beanshell.

Beanshell

Die Beanshell kann in reinem Java programmiert werden, wobei der hauptsächliche Vorteil darin liegt, dass Java vermutlich von den meisten Entwicklern beherrscht wird und somit keine Einarbeitungszeit anfällt. Beanshell unterstützt jedoch auch Features, die man aus der Skriptsprachenwelt kennt, wie z.B. dynamische Typisierung, Closures und prototypbasierte Objektorientierung. Insgesamt lässt sich damit ein kompakter und somit übersichtlicher Code schreiben.

BeanShell-Objekte sind reine Java-Objekte, die sich in fast allen Punkten wie reguläre Java-Objekte verhalten. Somit bestehen in der BeanShell die gleichen Möglichkeiten, welche auch in einem Anwendungs-Plugin geboten werden, allerdings mit kleinen Einschränkungen, auf die hier an dieser Stelle nicht weiter eingegangen wird.

Beanshell ist seit 2005 nicht mehr in der aktiven Entwicklung, was mit seiner Komplettheit und Robustheit erklärt wird. ([BSH], Lizenz: SPL und LGPL). Seit 2007 gibt es mit Beanshell2 einen Fork ([BSH2], Lizenz LGPL), welcher noch aktiv weiterentwickelt wird. Im Wesentlichen wird hier auf die Anpassung an neuere Java-Sprach-Features eingegangen. Ansonsten gibt es wenig Änderungen.

Des Weiteren existiert ein Editor-Plugin für NetBeans, welches aber leider noch nicht sehr ausgereift ist. Es gibt lediglich Syntax Coloring, aber keine Code-Vervollständigung.

Erweiterte Sprachfeatures gegenüber Java

BeanShell erweitert Java um zahlreiche Fähigkeiten, wie man sie von anderen Skript-

sprachen kennt. Im Folgenden wird ein kurzer Überblick über die Sprach-Features gegeben. Detailliertere Informationen können der BeanShell-Dokumentation [BSHdocs] entnommen werden.

Scopes

Scopes verhalten sich in BeanShell etwas anders als in Java. Werden Variablen in Blöken definiert gibt es zwei Unterschiede. Statisch typisierte Variablen (z.B. `String string = "foo";`) sind, wie in Java, nur in diesem Block sichtbar. Dynamisch typisierte Variablen (z.B. `string = "foo";`) sind auch außerhalb des Blockes sichtbar. Das ist z.B. bei der Verwendung von try-catch-Blöcken oder while-Schleifen von Vorteil.

Man kann auf Variablen außerhalb des eigenen Scopes zugreifen. Mit `super` gelangt man einen Ebene höher, mit `global` in die oberste Ebene des Skripts.

Interfaces

Es müssen nicht zwingend alle Methoden eines Interfaces implementiert werden. Das macht sich ggf. durch einen Laufzeitfehler bemerkbar.

Dynamische Argumente

Argumente von Methoden bzw. Funktionen können dynamisch sein:

```
int add(value1, value2) { return a+b; }
```

Das setzt natürlich vorraus, dass sich die verwendeten Typen addieren lassen. Zahlen-Typen werden z.B. addiert und Strings konkateniert.

Exceptions können in `catch`-Blöcken untypisiert gefangen werden: `catch (ex) {...}`

Verschachtelte Methoden/Funktionen

Eine Methode/Funktion kann weitere Funktionen enthalten. Das ist im Zusammenhang mit den Closures von Nutzen.

Closures

Methoden/Funktionen können `this` zurückgeben. Dies erlaubt es Methoden/Funktionen als Parameter an andere Methoden/Funktionen zu übergeben. Außerdem hat man Zugriff auf die inneren Methoden/Funktionen und Daten.

Vereinfachte Syntax

Für Getter bzw. Setter gibt es vereinfachte Zugriffe. Wenn z.B. das Objekt `person` die Methoden `getName()` und `setName(String name)` hat, kann man darauf wie folgt zugreifen:

```
person.name = "Kalle";                  // anstelle von person.setName("Kalle")
println("Name: " + person.name);        // anstelle von ... + person.getName()
```

Das setzt allerdings die Einhaltung der Namenskonventionen vorraus, wie sie z.B. für JavaBeans vorgeschrieben sind.

switch-case

switch-case-Anweisungen können auch mit Objekten benutzt werden.

Imports

Imports können an jeder Stelle des Codes auftauchen, wie man es z.B. von Scala kennt.

Einschränkungen und Nachteile der BeanShell

Da das Methoden-Dispatching zur Laufzeit via Reflections aufgelöst wird, sind Methodenaufrufe in der BeanShell zeitintensiver als in der reinen Java-Umgebung der Anwendung. Wenn in einem Test sehr viele Methodenaufrufe passieren, wäre zu überlegen, ob der betreffende Code nicht besser in einem Modul aufgehoben wäre. Dieses Modul könnte dann eventuell in einem Skript benutzt werden.

4 Das modulare Testsystem Sopas TT

Anhand der Designentscheidungen des vorherigen Kapitels wird nun das modulare Testsystem Sopas TT entwickelt. Der Name Sopas TT leitet sich von einer Software ab, die bei SICK entwickelt wurde: Sopas ET – Sopas Engineering Tool. In Anlehnung daran, steht TT hier für Testing Tool.

Sopas ist der Name des Netzwerkprotokolls der SICK-Sensoren. Auch wenn später in der Hauptsache Sopas-Geräte getestet werden sollen, ist Sopas TT nicht auf dieses Protokoll beschränkt. In der in diesem Kapitel beschriebenen Beispielimplementation kommt eine rudimentäre Modbus/TCP-Implementation (siehe Abschnitt 4.4 BitIO) zum Einsatz.

Ziel der gegenwärtigen Implementation ist eine Anwendung, die das Skripten eines Wizards und eines Geräts demonstriert. Die GUI der Anwendung besteht lediglich aus dem Hauptfenster der Anwendung, einem Ausgabe-View als Tab, einer Toolbar und dem Wizard.

NetBeans Platform unterstützt sowohl die eigenen NetBeans Platform Modules als auch OSGi Bundles, wie sie u.a. auch von Eclipse unterstützt werden. In diesem Projekt basieren alle Module auf den NetBeans Platform Modules.

4.1 Main-Modul

Das Main-Modul ist der Startpunkt der Anwendung. Bisher konzentriert es sich darauf eine Action mit einem Tastenkürzel zu verknüpfen, es in das Programm-Menü einzuklinken und ein Toolbar-Icon zu setzen. So kann man auf drei Wegen ein Skript ausführen. Die Action öffnet einen Dateidialog, in dem man eine Beanshell-Datei auswählen kann, die dann dem Beanshell-Interpreter übergeben wird.

Neben dem auszuführenden Skript werden dem Interpreter noch zwei Objekte übergeben, auf die das Skript dann Zugriff hat:

Beim Ersten handelt es sich dabei um einen OutputWriter, eine standard Netbeans-Platform-Ausgabe, zu erreichen über das Menü "Window –> Output –> Output". Dorthin können vom Skript Kontrollausgaben geschrieben werden können. Es ist im Skript über den Namen "out" zu erreichen.

Beim zweiten Objekt handelt es sich um einen ProtocolWriter (siehe 4.5 ProtocolWriter). Hier kann man aus dem Skript heraus ein Prüfprotokoll schreiben. Erreichbar ist es unter dem Namen "protocol".

Diese beiden Objekte dienen der Vereinfachung, sie könnten auch im Skript instanziiert werden.

4.2 ScriptableWizard

ScriptableWizard ist ein Factory-Objekt, mit dessen Hilfe in einem Skript leicht ein Wizard gebaut werden kann.

Normalerweise muss bei einem NetBeans Platform Wizard für jedes Panel eine GUI erzeugt werden, was normalerweise mit dem GUI-Builder Matisse gemacht wird. Ferner gehört zu jedem Panel ein Controller, der die Eingaben verarbeitet und welcher manuell programmiert werden muss. Diese Herangehensweise eignet sich nicht für die Benutzung in einem Skript, da dies einen zu hohen Schreibaufwand bedeutet und fehleranfällig ist.

Die Besonderheit an ScriptableWizard ist, dass die GUI-Panels und die dazugehörigen Controller nicht statisch je *n* mal erzeugt werden. Statt dessen gibt es je einen Standard-Controller und eine Standard-GUI. Die Panels werden im Skript mit den gewünschten Swing-Elementen befüllt, die automatisch an das Standard-Layout angepasst werden.

Die folgenden Codeabschnitte sind Beispielhaft und dienen der Verdeutlichung, wie der Wizard benutzt wird.

Ein BeanShell-Skript beginnt mit den nötigen Imports, danach folgt die Methode `start()`, in der zunächst eine Wizard-Factory erzeugt und konfiguriert wird.

```
import de.sick.sopastt.scriptablewizard.ScriptableWizard;
import de.sick.sopastt.scriptablewizard.WizardPanel;
import de.sick.sopastt.scriptablewizard.WizardWorker;
import de.sick.sopastt.protocolwriter.ProtocolWriter;
import org.openide.WizardDescriptor;
import org.openide.NotifyDescriptor;
import org.openide.DialogDisplayer;

start() {
    // ----- create wizard factory and configure -----
    wizardFactory = new ScriptableWizard();
    wizardFactory.configure("Show Wizard Panel Widgets");
```

Dieser Factory können nun Panels hinzugefügt werden. Über die zurückgegebene Referenz können dem Panel Swing-Elemente hinzugefügt werden.

```
    // ----- 0. widget demo panel -----
    panel = wizardFactory.addPanel("Widget-Demo");
    panel.addText("Dies ist ein Textfeld. Hier drunter ist eine Trennlinie."
        + " Bislang koennen die Widgets nur in einer Spalte untereinander stehen.");
    panel.addDivider();
    panel.addInputField("inputField1", "Eingabefeld", "", 1, false);
    panel.addInputField("inputField2", "Eingabepflichtfeld",
        "Text- und Input-Widgets koennen mehrere Zeilen haben.", 3, true);
    panel.addDivider();
    panel.addCheckBox("checkbox", "Es gibt Checkboxen", false);
    panel.addDivider();
```

```
panel.addText("Und es gibt Radio Buttons (es koennen auch mehrere Button Gruppen"
    + " auf einem Panel sein)");
panel.addSpacer();
panel.addRadioButtons("radioButtons", -1, new String[] {"Button 1", "Button 2"});
panel.addDivider();
panel.addText("Diese Meldungen hier unten werden fuer Radio Buttons"
    + " und Pflichtfelder automatisch vom Wizard erzeugt.");

// ----- 1. finish panel -----
panel = wizardFactory.addPanel("Ende");
panel.addText("Wenn die Checkbox gesetzt war, gibt das Skript nach Beenden"
    + " des Wizards noch die Eingabedaten auf der"
    + " Netbeans-Ausgabe aus (Menue Window -> Output -> Output).");
```

Nach dem Starten des Wizards würde man einen Wizard erhalten wie er in Abbildung 1 zu sehen ist.

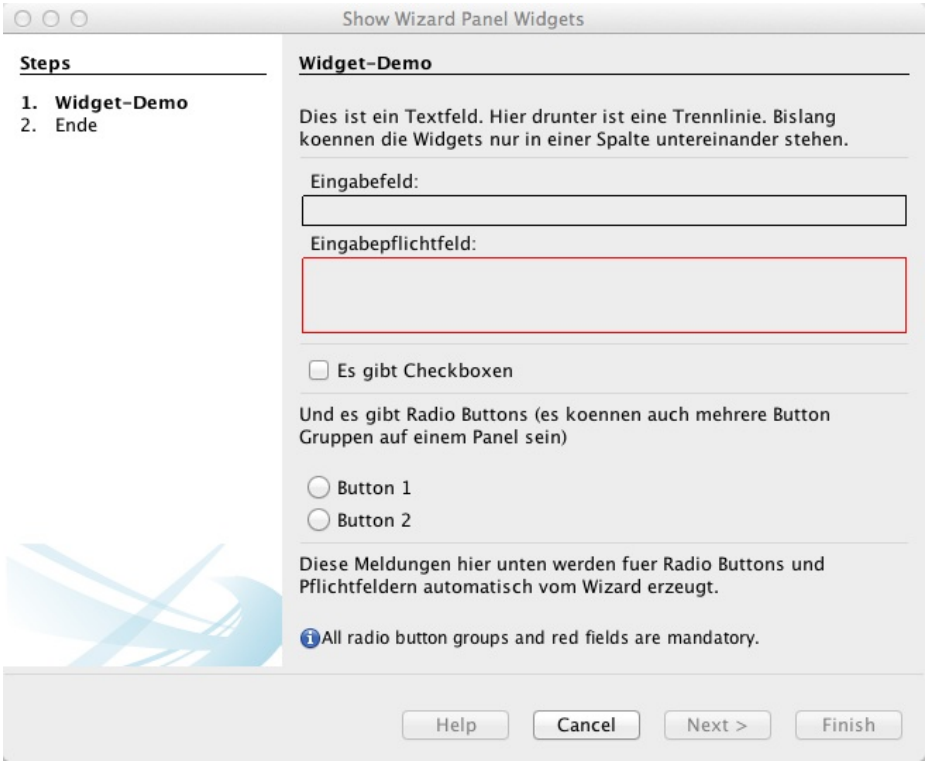

Abbildung 1: Beispiele für Swing-Elemente in einem Panel

Neben der nötigen Layout-Anpassung übernimmt der Wizard für den Nutzer die Verriegelung der Next- und Finish-Buttons. Bei Pflichteingabefeldern und bei Radio Buttons überprüft der Wizard, genauer das WizardPanel, ob die Bedingungen erfüllt sind und gibt dann erst die Buttons frei. Bei Nichterfüllung sind sie ausgegraut und es wird ein Hinweistext eingeblendet.

Beim Panel-Wechsel werden die eingegebenen Daten in den Key-Value-Store des Wizards geschrieben. Der Key ist der Name des jeweiligen Widgets, den man bei der Erzeugung vergeben hat. Auf diese Daten kann vom Skript aus zugegriffen werden. Nach dem Beenden des Wizards kann dieser nicht erneut verwendet werden. Auf die ange-

legten Daten im Key-Value-Store kann weiterhin zugegriffen werden.

Neben den Panels benötigt der Wizard noch einen WizardWorker. Das ist eine abstrakte Klasse, die durch die Methode `execute(WizardPanel panel)` definiert ist. Diese Methode wird nach jedem Klick auf Next oder Finish ausgeführt. Hier ein Beispiel für einen WizardWorker:

```
wizardFactory.setWizardWorker(new WizardWorker() {
    public void execute(WizardPanel panel) {
        switch (panel.getPanelNumber()) {
            case 0: // start panel
                    break;
            case 1: // user details panel
                    break;
            case 2: // final panel
                    break;
        }
    }
});
```

Anhand des übergebenen WizardPanels kann man das aktuellen Panel erfragen und entsprechend verfahren. Einzelne Statements kann man in den jeweiligen Case-Block einfügen oder Funktionen aufrufen.

Nachdem die Wizard-Factory alle Informationen erhalten hat, kann man den Wizard erzeugen lassen und ihn aufrufen. In dem folgenden Beispiel wird nach der Beendigung des Wizards das Protokoll gesichert, sofern die Checkbox "protocolSave" aktiviert wurde.

```
wizard = wizardFactory.buildWizard();

if (DialogDisplayer.getDefault().notify(wizard) == WizardDescriptor.FINISH_OPTION) {
    if (wizard.getProperty("protocolSave") == true) {
        fileChooser = new JFileChooser();
        fileChooser.setFileFilter(
                        new FileNameExtensionFilter("Textdateien", new String[] ));
        if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
            file = fileChooser.getSelectedFile();
            protocol.saveAsText(file, true);
        }
    }
}
```

Zusammengefasst muss man folgende Punkte bei der Erstellung eines Wizards beachten:

1. Wizard-Factory (ScriptableWizard) erzeugen
2. Wizard-Factory konfigurieren
3. WizardWorker an Wizard-Factory übergeben. Einzige zu implementierende Methode ist `execute(WizardPanel panel)`. In dieser kann via switch-case anhand der Panel-Nummer verzweigt werden.
4. Panel hinzufügen. Als Rückgabe erhält man eine Referenz auf das Panel, damit es mit UI-Elementen bestückt werden kann.

5 Swing-Widgets zu Panel hinzufügen.

6 (Die Schritte 4 und 5 so oft wiederholen, wie Panels benötigt werden.)

7 Wizard von Wizard-Factory erzeugen lassen.

8 Wizard ausführen.

9 Nach Beendigung des Wizards kann je nach Beendigungsstatus (Abgebrochen, Erfolgreich beendet) verfahren werden.

4.3 BeanShell

BeanShell ist lediglich ein Modul-Wrapper um die BeanShell-Jars. Dies vereinfacht nachliefern von BeanShell-Updates.

Um die BeanShell zu nutzen muss lediglich ein Interpreter instanziiert werden, sowie die Beanshell-Datei geladen, geparst und ausgeführt werden.:

```
Interpreter shell = new Interpreter();  // Interpreter anlegen
shell.source("/pfad/datei.bsh");        // Shell-Datei laden und parsen
shell.eval("start()");                  // Ausführen
```

Eine BeanShell-Datei kann einfach nur Befehle enthalten. Hier reicht es `eval()` ohne Parameter aufzurufen. In einer BeanShell-Datei können aber auch Funktionen definiert werden, die als Einstieg dienen, wie im obigen Beispiel gezeigt. Das Interpreter-Objekt bleibt nach einem Aufruf noch gültig und kann mehrfach aufgerufen werden. Man kann aus der Java-Umgebung heraus Objekte an den Interpreter übergeben und auch den Interpreter nach Objekten befragen:

```
OutputWriter writer = IOProvider.getDefault().getIO("Output", false).getOut();
shell.set("out", writer);           // writer als out der Shell zur Verfügung stellen
shell.eval("start()");              // Ausführen
Object obj = shell.get("foo");      // nach dem Object foo fragen
```

In diesem Beispiel wird ein OutputWriter angelegt. Ist schon eine Instanz mit dem Namen "Output" vorhanden, wird diese zurückgegeben, andernfalls wird eine neue angelegt. Dessen Referenz wird dann an die Shell übergeben, von wo aus sie dann über den Namen "out" ansprechbar ist. Am Ende wird die Shell gefragt, ob es ein Objekt mit dem Namen "foo" gibt. Wenn ja, bekommt man die Objektreferenz, andernfalls `null`.

4.4 BitIO

BitIO ist eine Netzwerk-IO-Box mit je 8 diskreten Ein- und Ausgängen. Die Kommunikation mit der Box läuft über Modbus/TCP. BitIO ist auch der Name des Moduls für Sopas TT, was eine rudimentäre Implementation des Protokolls darstellt. Es wird derzeit das Lesen von Eingängen und Schreiben von Ausgängen unterstützt.

```
public int readDiscreteInput(int register, int count, boolean unsigned)
```

Diese Methode liest `count` zusammenhängende Bits ab Adresse `register`, wobei das Ergebnis-Byte wahlweise signed oder unsigned interpretiert wird.

```
public void writeDiscreteOutput(int register, int count, int value)
```

Hier wird ein Bitmuster in die gewünschte Registeradresse geschrieben. Auch hier gibt `count` die Anzahl der zusammenhängenden Ausgänge an.

Die erforderlichen Parameter hängen vom verwendeten Modbus-Geräts ab. Die aktuelle Implementation unterstützt noch keine Modbus-Geräte mit mehr als 31 diskreten Ein- bzw. Ausgängen und noch keine maskierten Zugriffe.

4.5 ProtocolWriter

Dieses Modul besteht aus einer einzigen Klasse. Es soll am Ende ein einfaches Protokoll in ASCII ausgeben. Ein Protokoll besteht aus drei Teilen. Als erstes sind das die Metadaten, wie z.B. Name des Tests, Name des Testers, Testdatum, Notizen, Randbemerkungen. Diese Daten sind von der Klasse nicht vorgegeben und werden als generische Daten in einem Key-Value-Store gehalten.

```
public <T> void setInfo(String name, T value)
public <T> T getInfo(String name)
public <T> removeInfo(String name)
public HashMap<String, Value<?>> getInfosAsHashMap()
public ArrayList<String> getInfosAsArrayList()
public String getInfosAsString()
public boolean getTestResult()
```

Der darauf folgende Teil ist das eigentliche Protokoll. In diesem sind die Ergebnisse der Teiltests und das Gesamtergebnis zu finden. Diese Einträge werden automatisch zusätzlich ins Logbuch, welches Teil des Protokolls ist, eingetragen, damit man eventuelle Vorfälle leichter wiederfinden kann. Das Gesamtergebnis wird automatisch aus den Teilergebnissen ermittelt. Es müssen alle Teiltests erfolgreich sein, damit das Gesamtergebnis erfolgreich ist.

```
public void add(boolean success, String entry)
public ArrayList<String> getProtocolAsArrayList()
public String getProtocolAsString()
public int countOfProtocolEntries()
```

Das Logbuch dient dazu, diverse Kontrollausgaben aufzunehmen. Hier können Info-Ausgaben, Fehlermeldungen usw. eingetragen werden. Die Einträge bekommen automatisch einen Zeitstempel. Das Logbuch kann optional mit dem Protokoll abgespeichert werden.

```
public void log(String entry)
public ArrayList<String> getLogAsArrayList()
public String getLogAsString()
public int countOfLogEntries()
```

Diese Methode speichert das Protokoll in eine Text-Datei ab.

```
public boolean saveAsText(File file, boolean withLog)
```

Nachfolgend ein Beispiel für ein Protokoll. Die Rahmendaten speisen sich aus den Metadaten, die u.a. vom Wizard abgefragt werden. Der Protokoll-Teil enthält nur das Gesamtresultat und die Ergebnisse der Einzeltests. Die Logdaten sind optional und werden auf Wunsch mitgesichert. Sie können u.U. bei fehlgeschlagenen Tests bei der Analyse hilfreich sein.

```
Rahmendaten:

Test: Einfacher Kabeltest
Abschliessende Bemerkungen: -keine-
Datum: Thu Apr 26 13:07:43 CEST 2012
Notizen: Lorem ipsum...
Pruefer: Mike Brasch

----------------------------------------

Protokoll:

Entire Test Result: FAILED

Count up pattern test. : FAILED
Random pattern test. : passed

----------------------------------------

Log:

2012-04-26 13:07:45.686 : Successfully written/read 0 to/from device.
2012-04-26 13:07:45.712 : Successfully written/read 2 to/from device.
...
2012-04-26 13:07:47.088 : Successfully written/read 110 to/from device.
2012-04-26 13:07:47.113 : Successfully written/read 112 to/from device.
2012-04-26 13:07:49.129 : Reading 114 from device failed: Read timed out
2012-04-26 13:07:51.146 : Reading 116 from device failed: Read timed out
2012-04-26 13:07:51.163 : Reading 118 from device failed: Wrong transactionID (115, but
expected 117).
2012-04-26 13:07:51.206 : Successfully written/read 120 to/from device.
2012-04-26 13:07:51.233 : Successfully written/read 122 to/from device.
...
2012-04-26 13:07:52.916 : Successfully written/read 252 to/from device.
2012-04-26 13:07:52.941 : Successfully written/read 254 to/from device.
2012-04-26 13:07:52.941 : *** Count up pattern test. : FAILED ***
2012-04-26 13:07:52.970 : Successfully written/read 186 to/from device.
2012-04-26 13:07:52.997 : Successfully written/read 24 to/from device.
...
2012-04-26 13:07:58.058 : Successfully written/read 70 to/from device.
2012-04-26 13:07:58.083 : Successfully written/read 174 to/from device.
2012-04-26 13:07:58.084 : *** Random pattern test. : passed ***
```

5 Testablauf

Nach dem ersten Start der Anwendung präsentiert sich die Anwendung mit einem leeren Hauptfenster und einer Toolbar. Über das Menü "Window -> Output -> Output" oder CTRL+F4 öffnet sich das Ausgabefenster, in dem man sich Informationen für Diagnosezwecke ausgeben lassen kann. Über "Tools -> BeanShell-Skript ausführen" oder F1 oder einem Toolbar-Symbol kann ein Datei-Öffnen-Dialog geöffnet und ein BeanShell-Skript ausgewählt werden. Nach dem Laden des Skripts wird dieses sofort ausgeführt.

Abbildung 2 zeigt ein Remote-IO-Modul. Dieses Modell besitzt je 8 diskrete Ein- und Ausgänge, die über das Netzwerk abgefragt bzw. geschaltet werden können. Dieses Gerät ist die Basis der beiden folgenden, exemplarischen Tests.

Abbildung 2: Remote-IO-Module (BitIO)

5.1 Wizard-unterstützter automatischer Test

Zur Demonstration des automatischen Tests dient ein einfacher Kabeltest. Der in Abbildung 3 abgebildete Kabeltester wird an das Remote-IO-Modul angeschlossen. Die Kabel sollen auf Durchgang der einzelnen Adern getestet werden. Die Box besteht im Wesentlichen aus einem Inverter-Board und zwei Sub-D-Buchsen. Das Inverter-Board soll aus den Ausgangssignalen Hi-Pegel erzeugen, die über das zu testende Kabel auf die Eingänge geschaltet werden. Bei einem intakten Kabel soll das Eingangs-Bitmuster dem Ausgangs-Bitmuster entprechen.

An den außenliegenden Sub-D-Steckern wird das zu testende Kabel angeschlossen. Normalerweise würde dies über Adapter geschehen, aber für diesen Test wurde ein spezielles Kabel gebaut, bei dem man über den Dip-Schalter einzelne Adern ausschalten kann, um dadurch einen Kabelbruch zu simulieren.

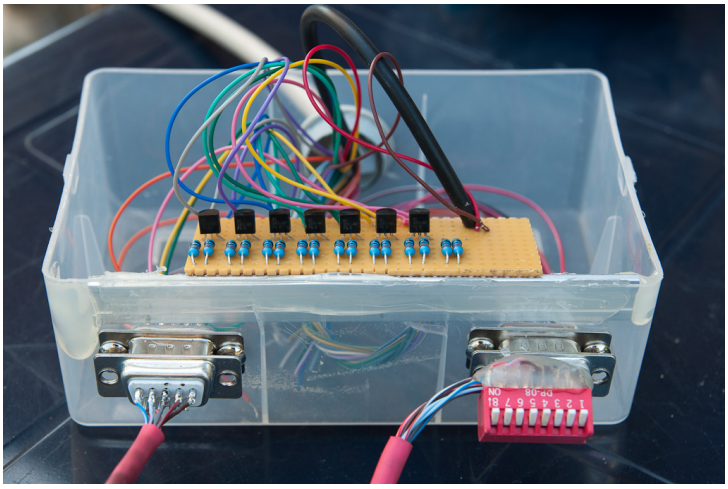

Abbildung 3: Kabeltester-Box

Im folgenden wird ein typischer Testlauf beschrieben. Der Quellcode des BeanShell-Skriptes zu diesem Test ist im Anhang unter "BeanShell-Skript: Kabeltester" zu finden.

Im Allgemeinen sollte die Startseite eines Wizards den oder die Tests kurz beschreiben und ggf. eine vorsichtige Zeitabschätzung geben, so wie hier (Abbildung 4) zu sehen.

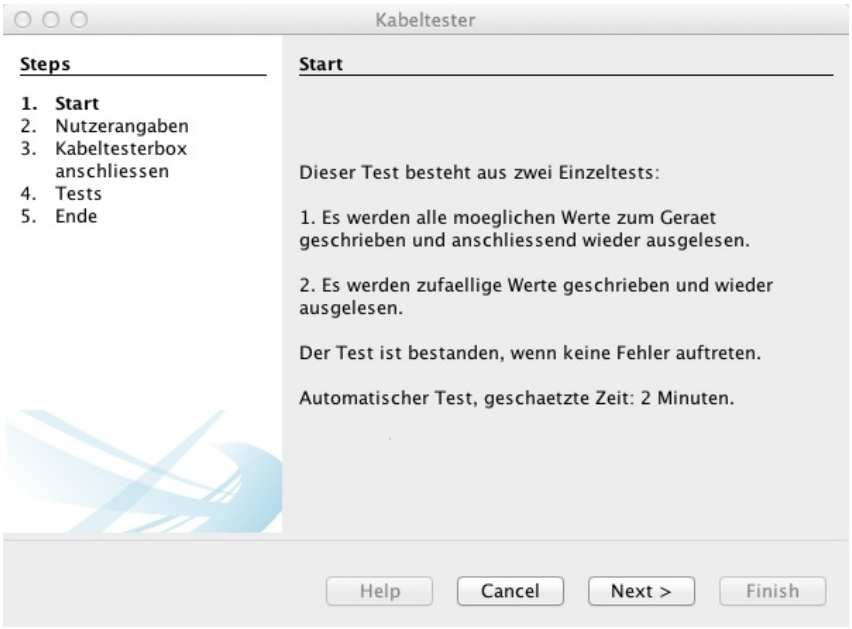

Abbildung 4: Wizard – Startseite mit Testbeschreibung

Als nächstes folgt das Panel mit den Nutzerangaben. Die Angaben sind nicht auf die in Abbildung 5 gezeigten festgelegt. Welche Daten erhoben werden obliegt dem Skript-Entwickler. Das Datum wird in diesem Beispiel automatisch ins Protokoll eingetragen. Sind Angaben unbedingt erforderlich, wird ihr Fehlen im unteren Teil des Fensters angezeigt und ein Weiterschalten zum nächsten Panel unterbunden.

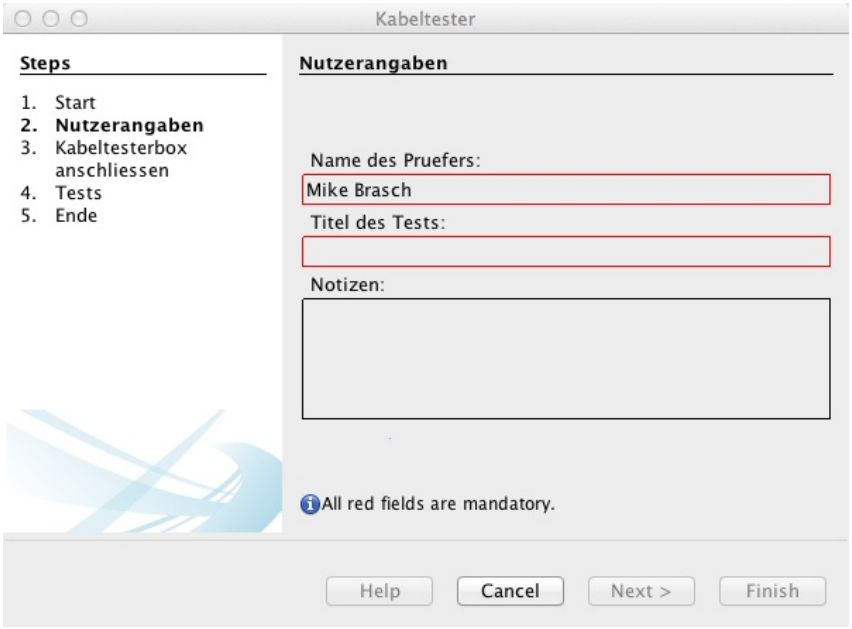

Abbildung 5: Wizard – Angaben zum Test

Das folgende Panel gibt Anweisungen zum Anschluß der Kabeltesterbox (Abbildung 6).

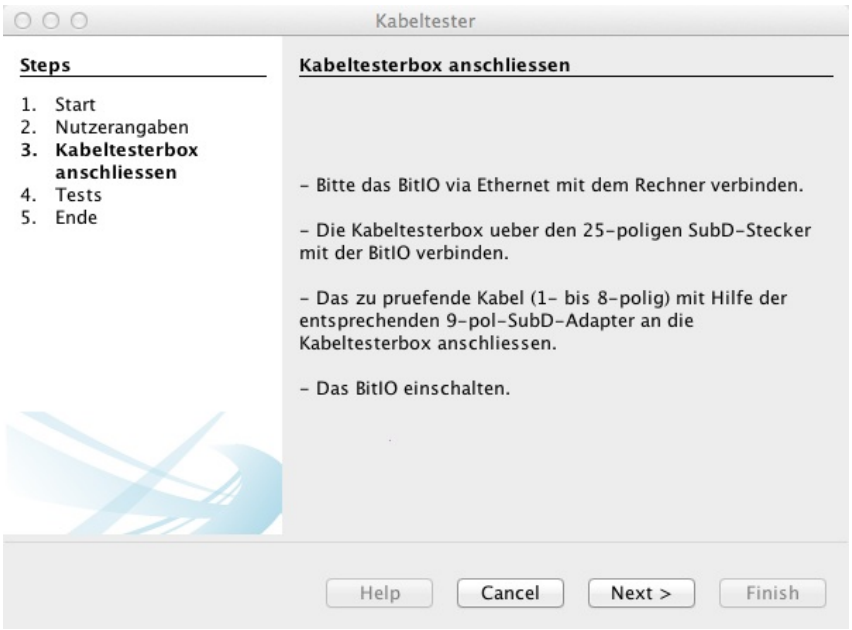

Abbildung 6: Wizard – Anschlußanweisungen

Nachfolgend erscheint das Panel, in dem der Fortschritt der Tests angezeigt wird (siehe Abbildung 7).

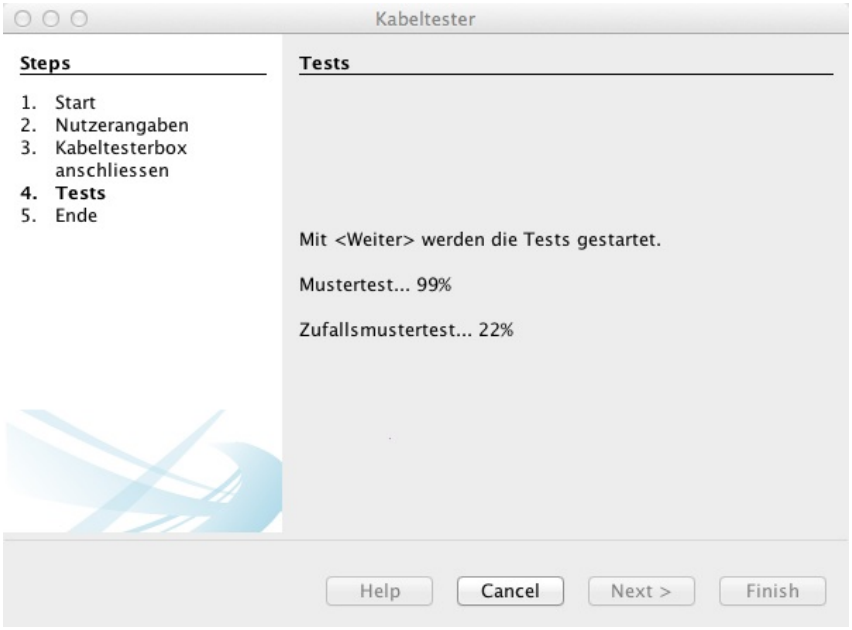

Abbildung 7: Wizard – Testfortschritt

Im letzten Panel kann man noch abschliessende Bemerkungen zum Protokoll hinzufügen, wie z.B. mögliche Gründe für das Scheitern eines Tests, oder sonstige, potentiell interessanten Ereignisse. Im Info-Bereich ist das Gesamttestresultat zu sehen. Hier kann der Testende angeben, ob nach Beendigung des Wizards das Protokoll gesichert und angezeigt werden soll (Abbildung 8).

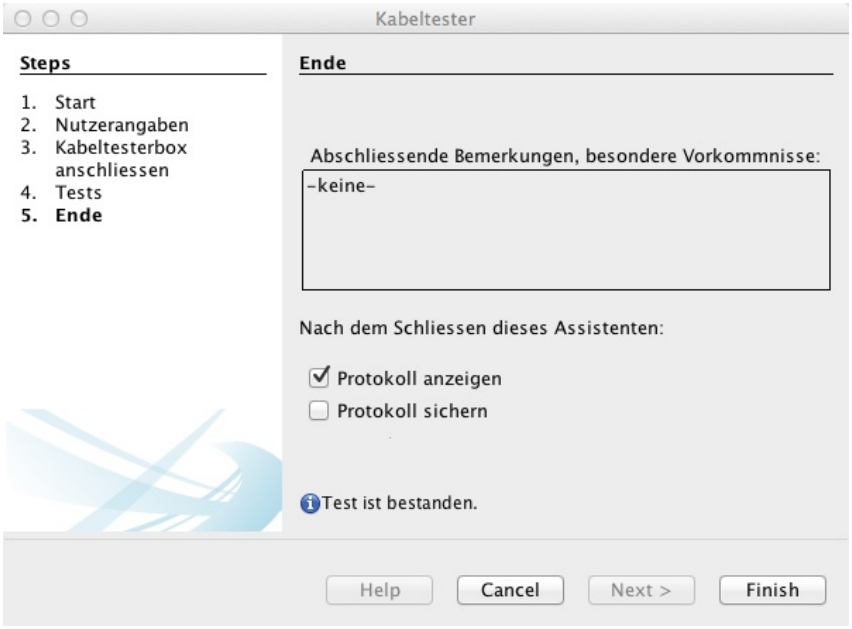

Abbildung 8: Wizard – Schlußseite, abschliessende Bemerkungen

5.2 Wizard-unterstützter manueller Test

Zur Demonstration dieses Tests dient die in Abbildung 9 dargestellte Platine, bestückt mit einem Taster und einer LED. Damit soll ein Test simuliert werden, bei dem der Test nicht automatisch ein Resultat ermittelt, sondern der Testende per Beobachtung das Resultat ermittelt und dem Wizard mitteilt.

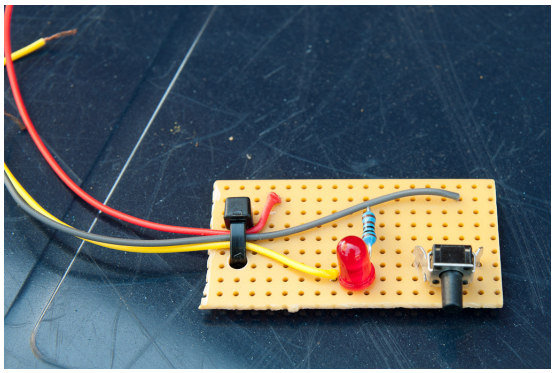

Abbildung 9: Platine mit Taster und LED

Der Test startet wie der bereits beschriebene automatische Test. Der Einweisungsseite folgt auch hier zunächst der Dialog zur Benutzerdateneingabe und auch die Anweisungen zum Anschluß an das Network-IO-Module. Ebenso fällt die Schlussseite dementsprechend aus. Nachfolgend die Panels, die typisch für den manuellen Test sind.

Wie in Abbildung 10 gezeigt erfolgt zunächst die Handlungsanweisung an den Testenden.

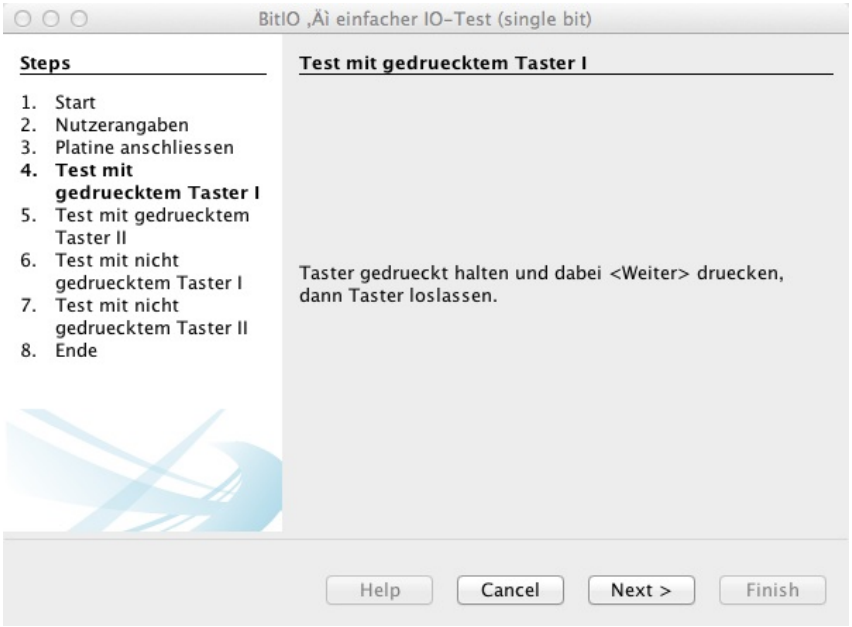

Abbildung 10: Wizard – Handlungsanweisungen

Nachdem der Test beendet wurde, wird der Tester aufgefordert, seine Beobachtung anzugeben, so wie in Abbildung 11 dargestellt.

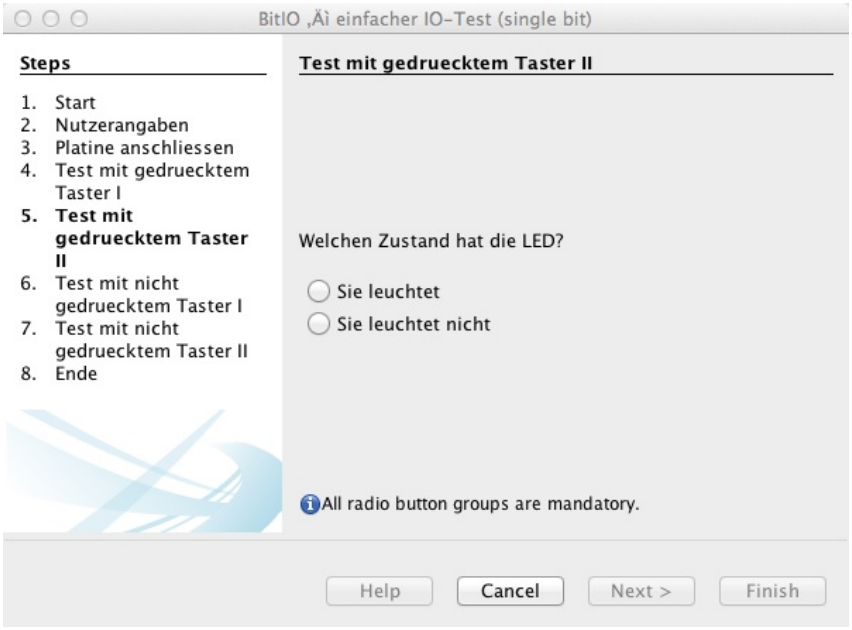

Abbildung 11: Wizard – Abfrage

6 Fazit und Ausblick

Entsprechend den Anforderungen aus Kapitel 2 hat sich die Designentscheidung zugunsten der NetBeans Platform als passende Wahl herausgestellt. Mit Hilfe der BeanShell war es möglich die automatisierte Testdurchführung zu skripten. Die implementierten Module der Sopas TT Testumgebung ermöglichen die beschriebene automatische und manuelle Testdurchführung. Abschliessend werden mögliche Erweiterungen beschrieben.

Für eine weitere Entwicklung des Projekts könnten folgende Punkte ausgearbeitet werden:

Der ScriptableWizard kann bislang Tests seriell in der programmierten Folge abarbeiten. Es ist derzeit nicht möglich, den Testlauf vorzeitig abzubrechen und zum Ergebnis-Panel zu springen. Ebenso sind bisher keine Verzweigungen möglich. Es wurde bislang auf Rückwärtssprünge verzichtet, zugunsten einfacherer Scripte. Solange diese fehlenden Funktionen optional sein können, wäre es vorteilhaft sie einzubauen, um komplexere Tests mit dem Wizard abbilden zu können.

Um mit dem Gros der SICK-Sensoren kommunizieren zu können, muss das Sopas-Protokoll noch implementiert werden. Einige SICK-Projekte benutzen mit der Sopas Communication Library SCL eine Java-Bibliothek, die für dieses Projekt geeignet scheint.

Des Weiteren spricht einiges für die Weiterentwicklung der GUI. Hier würde sich ein Projekt- und Modul-Manager anbieten, mit dessen Hilfe man die anwendungsspezifischen Module (Geräte, Visualisierungen etc.) verwalten und Projekte grafisch bearbeiten kann. Auch ein Editor, in dem man Skripte schreiben und ggf. Testen kann, wäre hilfreich. Ferner wären Module denkbar, die Sensordaten grafisch oder statistisch aufbereiten. Diese könnten geskriptete Tests bei hochperformanten Aufgaben unterstützen.

Der ProtocolWriter könnte zu einem Interface umgebaut werden. Aus dem derzeitigen ProtocolWriter könnte dann ein TextProtocolWriter werden. Entsprechend dazu könnten dann weitere ProtocolWriter geschrieben werden, wie z.B. einen XMLProtocolWriter für maschinell weiterverarbeitbare Protokolle oder einem PDFProtocollWriter, der formatierte Protokolle erzeugt.

Literaturverzeichnis

[NBP] NetBeans Platform, Zugriffsdatum: 1.5.2012
http://platform.netbeans.org

[BSH] BeanShell, Zugriffsdatum: 12.4.2012
http://beanshell.org

[BSHdocs] BeanShell Docs, Zugriffsdatum: 1.5.2012
http://beanshell.org/docs.html

[BSH2] BeanShell2, Zugriffsdatum: 12.4.2012
http://code.google.com/p/beanshell2

Anhang

1 BeanShell-Skript: Kabeltester

```
import de.sick.sopastt.bitio.BitIO;
import de.sick.sopastt.scriptablewizard.SkriptableWizard;
import de.sick.sopastt.scriptablewizard.WizardPanel;
import de.sick.sopastt.scriptablewizard.WizardWorker;
import de.sick.sopastt.protocolwriter.ProtocolWriter;
import java.io.File;
import java.net.UnknownHostException;
import java.util.Date;
import java.util.Random;
import javax.swing.*;
import javax.swing.JFileChooser;
import javax.swing.filechooser.FileNameExtensionFilter;
import org.openide.util.Exceptions;
import org.openide.util.Lookup;
import org.openide.windows.IOProvider;
import org.openide.windows.OutputWriter;
import org.openide.WizardDescriptor;
import org.openide.NotifyDescriptor;
import org.openide.DialogDisplayer;

// The calling environment sets the following objects for convenience:
// ********************************************************************
// OutputWriter     out         for debugging output to the applications output view.
// ProtocolWriter   protocol    for writing protocol

start() {
   wizardFactory = new ScriptableWizard();
   wizardFactory.configure("Kabeltester");

   // ----- create wizardFactory worker -----

   wizardFactory.setWizardWorker(new WizardWorker() {
      bitio = new BitIO();

      public void execute(WizardPanel panel) {
         switch (panel.getPanelNumber()) {
            case 0: // start panel
                  break;
            case 1: // user details panel
                  protocol.setInfo("Pruefer", wizard.getProperty("nameOfTester"));
                  protocol.setInfo("Test", wizard.getProperty("titleOfTest"));
                  protocol.setInfo("Notizen", wizard.getProperty("notice"));
                  protocol.setInfo("Datum", new Date());
                  break;
            case 2: // make connections panel
                  connect(panel);
```

```
                break;
        case 3: // test panel
                startTests(panel);
                break;
        case 4: // final panel
                protocol.setInfo("Abschliessende Bemerkungen", wizard.getProperty("summary"));
                if (bitio != null && bitio.isConnected()) { bitio.disconnect(); }
                break;
    }
}

public void connect(WizardPanel panel) {
    try {
        bitio.init("192.168.1.11", 502, 2000);
        bitio.connect();
    }
    catch (IOException ioe) {
        protocol.log("Connecting to device failed: " + ioe.getMessage());
        panel.setErrorMessage("Verbindung konnte nicht hergestellt werden.");
    }
}

public void startTests(WizardPanel panel) {
    testCountUpPattern(panel);
    testRandomPattern(panel, 200);

    if (protocol.getTestResult()) {
        panel.setInformationMessage("Test ist bestanden.");
    }
    else {
        panel.setInformationMessage("Test ist NICHT bestanden.");
    }
}

public void testCountUpPattern(WizardPanel panel) {
    text = "\nMustertest... ";
    textField = panel.addText(text);
    passed = true;
    int max = 255;

    for (i=0; i<max; i+=2) {
        textField.setText(text + (i*100/max) + "%");
        try {
            if (readWrite(i) == false) { passed = false; }
        }
        catch (SocketTimeoutException ste) {
            out.println("testCountUpPattern: SocketTimeoutException: " + ste.getMessage());
            passed = false;
            protocol.log("Reading " + i + " from device failed: " + ste.getMessage());
        }
        catch (IOException ioe) {
            out.println("testCountUpPattern: IOException: " + ioe.getMessage());
            passed = false;
            protocol.log("Reading " + i + " from device failed: " + ioe.getMessage());
        }
    }

    protocol.add(passed, "Count up pattern test.");
}

public void testRandomPattern(WizardPanel panel, int rounds) {
    text = "\nZufallsmustertest... ";
    textField = panel.addText(text);
```

```
        passed = true;
        random = new Random(1);
        int max = 127;

        for (i=0; i<=rounds; i++) {
           textField.setText(text + (i*100/rounds) + "%");
           try {
              if (readWrite(random.nextInt(max+1)*2) == false) { passed = false; }
           }
           catch (SocketTimeoutException ste) {
              out.println("testRandomPattern: SocketTimeoutException: " + ste.getMessage());
              passed = false;
              protocol.log("Reading " + i + " from device failed: " + ste.getMessage());
           }
           catch (IOException ioe) {
              out.println("testRandomPattern: IOException: " + ioe.getMessage());
              passed = false;
              protocol.log("Reading " + i + " from device failed: " + ioe.getMessage());
           }
        }

        protocol.add(passed, "Random pattern test.");
     }

     public boolean readWrite(int value) throws IOException, SocketTimeoutException {
        success = true;
        try { Thread.sleep(10); } catch (InterruptedException ie) {}

        bitio.writeDiscreteOutput(0, 8, value);

        try { Thread.sleep(10); } catch (InterruptedException ie) {}

        retValue = bitio.readDiscreteInput(0, 8, true);
        if (retValue != value) {
           success = false;
           protocol.log("Reading " + value + " from device failed: read " + retValue);
        }
        else {
           protocol.log("Successfully written/read " + value + " to/from device.");
        }

        return success;
     }

  });

  // ----- create panels ------

  // ----- 0. start panel -----

  panel = wizardFactory.addPanel("Start");
  panel.addText("Dieser Test besteht aus zwei Einzeltests:"
     + "\n\n1. Es werden alle moeglichen Werte zum Geraet geschrieben"
     + " und anschliessend wieder ausgelesen."
     + "\n\n2. Es werden zufaellige Werte geschrieben und wieder ausgelesen."
     + "\n\nDer Test ist bestanden, wenn keine Fehler auftreten."
     + "\n\nAutomatischer Test, geschaetzte Zeit: 2 Minuten.");

  // ----- 1. User details panel -----

  panel = wizardFactory.addPanel("Nutzerangaben");
  panel.addInputField("nameOfTester", "Name des Pruefers", "Mike Brasch", 1, true);
  panel.addInputField("titleOfTest", "Titel des Tests", "Lorem ipsum...", 1, true);
```

```
   panel.addInputField("notice", "Notizen", "", 5, false);

   // ----- 2. preparation panel -----

   panel = wizardFactory.addPanel("Kabeltesterbox anschliessen");
   panel.addText("- Bitte das BitIO via Ethernet mit dem Rechner verbinden."
      + "\n\n- Die Kabeltesterbox ueber den 25-poligen SubD-Stecker mit der BitIO verbinden."
      + "\n\n- Das zu pruefende Kabel (1- bis 8-polig) mit Hilfe der entsprechenden"
      + " 9-pol-SubD-Adapter an die Kabeltesterbox anschliessen."
      + "\n\n- Das BitIO einschalten.");

   // ----- 3. test panel -----

   panel = wizardFactory.addPanel("Tests");
   panel.addText("Mit <Weiter> werden die Tests gestartet.");

   // ----- 4. result panel -----

   panel = wizardFactory.addPanel("Ende");
   panel.addInputField("summary", "Abschliessende Bemerkungen, besondere Vorkommnisse",
      "-keine-", 5, false);
   panel.addSpacer();
   panel.addText("Nach dem Schliessen dieses Assistenten:");
   panel.addSpacer();
   panel.addCheckBox("protocolShow", "Protokoll anzeigen", true);
   panel.addCheckBox("protocolSave", "Protokoll sichern", false);

   // ----- run wizard -----

   wizard = wizardFactory.buildWizard();

   if (DialogDisplayer.getDefault().notify(wizard) == WizardDescriptor.FINISH_OPTION) {

      if ((Boolean)wizard.getProperty("protocolSave")) {
         fileChooser = new JFileChooser();
         fileChooser.setFileFilter(new FileNameExtensionFilter("Textdateien", new String[] ));

         if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
            file = fileChooser.getSelectedFile();
            protocol.saveAsText(file, true);
         }
      }

      if ((Boolean)wizard.getProperty("protocolShow")) {
         out.println(protocol.getInfosAsString());
         out.println(protocol.getProtocolAsString());
         out.println(protocol.getLogAsString());
      }

   }

   return 0;
}
```

Versicherung über Selbstständigkeit

Hiermit versichere ich, daß ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 2. Mai 2012

Ort, Datum

Unterschrift