# DISCOURSE 2004

## Programmierprojekt

### Aufgabe 1:

## Integration eines DCL-Frontends in Visual Studio .NET 2003

**von:**

**Jan Möller**
**Philipp Sommer**

# Inhaltsverzeichnis:

# 1. Aufgabenstellung und Einführung

Dieses Projekt entstand im Rahmen der DISCOURSE Block-Lehrveranstaltung. Die Aufgabenstellung dieser Arbeit war die Entwicklung eines Visual Studio .NET 2003 Add-ins für das *Distributed Control Lab* (DCL).

Das DCL bietet eine Infrastruktur um verschiedenste Experimente übers Internet durchführen zu können. Ein Benutzer ist in der Lage ein Steuerungsprogramm zu schreiben und dieses dann über eine Web-Schnittstelle an das DCL zu senden. Dieser so genannte Job wird anschließend kompiliert und ausgeführt. Der Benutzer kann über Kameras und Statusinformationen den Experimentverlauf verfolgen.
Wissenschaftlicher Hintergrund des DCLs ist die Frage, wie man ein voraussagbares Systemverhalten in einer instabilen Umgebung erreicht. Dinge wie Fehlertoleranz, Zeitverhalten, Sicherheit oder Ressourcenverbrauch spielen dabei eine zentrale Rolle.

Dieses Programmierprojekt zielte auf die Bedienung und Benutzung des DCLs ab. Zur Zeit existiert eine Webseite, über die Experimente gesteuert und Jobs verwaltet werden können. Unser Ziel war die Entwicklung eines Add-ins für Visual Studio .NET 2003. Alle Funktionen, die über die Webseite verfügbar sind, sollten möglichst in die Entwicklungsumgebung integriert werden.
Um dies zu erreichen, war zuerst die intensive Einarbeitung in die Schnittstellen von Visual Studio .NET 2003 nötig.

# 2. Erweiterungsmöglichkeiten von Visual Studio

Visual Studio bietet generell die Möglichkeit, Aktionen zu automatisieren oder die IDE (Integrated Development Environment) zu erweitern. Microsoft stellt für dieses Feature eine Programmierschnittstelle zur Verfügung, auch bekannt als Automation Model.

Das Automation Model enthält Objekte und Schnittstellen, die es einem Entwickler erlauben, programmatisch die Tool- oder Document Windows, wie den Code Editor, Solution Explorer oder die Tasklist zu kontrollieren oder zu automatisieren. Aber auch andere IDE Funktionen lassen sich steuern, wie zum Beispiel Solution und Project Build Configurations, das Editieren von Code oder auch das Debugging. Mit der Benutzung des Automation Models ist man also in der Lage VSMacros, Add-ins und Wizards zu erstellen, um Aufgaben zu automatisieren, die IDE zu manipulieren oder auch auf Events der IDE zu reagieren.

Die folgende Abbildung 1 zeigt das Automation Model im Überblick:

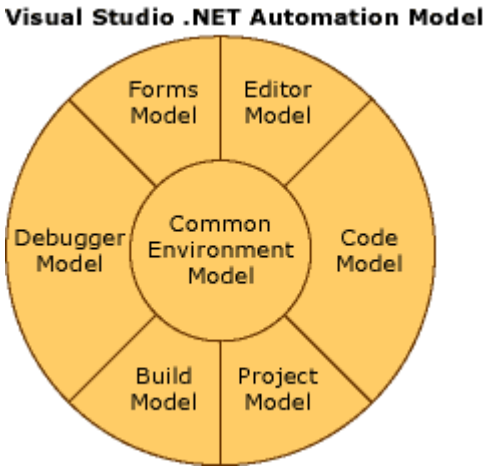


**Abbildung 1: Automation Model im Überblick**

Jedes dieser Modelle enthält Objekte mit denen man die spezifischen User-Interface-Elemente aber auch IDE Funktionalitäten steuern bzw. manipulieren kann.

Zusätzlich zur Unterteilung, wie in Abbildung 1, muss man noch betrachten, dass die in Visual Studio .NET 2003 integrierten Sprachen (C++ .NET, VB .NET, C# .NET) ihre eigenen Erweiterungen mitbringen können. Dies gilt insbesondere für das Build, Project und Code Model. Dennoch existieren gemeinsame Objekte, die für alle Sprachen gelten.

Das Spektrum für den Zugriff auf das Automation Model umfasst 3 Wege, die im folgenden erläutert werden.

## 2.1. VSMacros

Die VSMacros sind der einfachste Weg Visual Studio zu automatisieren oder die IDE zu erweitern. Sie lassen sich sehr effizient mit Hilfe der Aufzeichnungsfunktion erzeugen. So kann man Benutzeraktionen, die mehrere Kommandos benötigen, komfortabel und effizient in ein Kommando umwandeln. Doch nicht nur über die Aufzeichnungsfunktion können VSMacros erzeugt werden, sondern auch mit der Macros IDE. Sie enthält einen Editor und stellt die Möglichkeit des Debuggings zur Verfügung.
VSMacros eignen sich besonders für die Automatisierung von Aktionen, die mehrere Schritte umfassen. So kann man zum Beispiel Einstellungen unter Berücksichtigung von Kontextinformationen mit nur einer Tastenkombination oder einem Klick ändern. Den Automatisierungswünschen des Benutzers sind keine Grenzen gesetzt.

Mit VSMacros ist man also in der Lage, die Entwicklungsumgebung und die Kommandos an seine individuellen Bedürfnisse anzupassen.

## 2.2. Add-ins und Wizards

Add-ins sind richtige Applikationen, die die Funktionalität der IDE erweitern. Sie sind COM Objekte und implementieren die IDTExtensibility2-Schnittstelle. Die Kommunikation mit der IDE erfolgt durch das Automation Model, das in der EnvDTE type library (dte.olb) enthalten ist. Man hat demnach vollen Zugriff auf das Automation Model.
Die in Visual Studio integrierten Sprachen bieten zusätzliche Objekte. So stellt zum Beispiel Visual C++ .NET eigene Bibliotheken für das Code Model und das Projekt Model zur Verfügung, die beide die sprachspezifischen Erweiterungen enthalten.
Die Wahl der Programmiersprache bleibt dem Programmierer überlassen. Add-ins können mit COM - fähigen Sprachen wie zum Beispiel Visual C++ .NET, Visual Basic .NET oder Visual C# .NET programmiert werden.

Wenn man also neue Funktionen der IDE hinzufügen möchte, verwendet man Add-ins. Sie bieten mehr Funktionen als VSMacros. So kann man mit ihnen eigene Property Pages für den Options – Dialog hinzufügen, eigene Tool Windows erzeugen oder dynamisch Menüs und Toolbars manipulieren. Dies geht mit VSMacros nicht.

Wizards sind eine Serie von automatischen Schritten, die dem Benutzer helfen, komplexere Aufgaben zu lösen. Wizards werden zum Beispiel beim Anlegen eines Add-in Projekts oder Deployment Projekts benutzt.
Visual Studio .NET 2003 bietet die Möglichkeit eigene Wizards zu erzeugen, in dem man ein COM – Objekt erstellt und die IDTWizard-Schnittstelle implementiert. Sie werden häufig durch die New Projekt oder New File Dialogbox aufgerufen.

## 2.3. Visual Studio Integrator Program (VSIP)

Das VSIP wurde entwickelt, um Unternehmenslösungen wie zum Beispiel neue Programmiersprachen zur IDE hinzuzufügen. Dafür muss man einen neuen Projekttyp erzeugen, vielleicht einen benutzerdefinierten Editor oder die Debugging Funktionen erweitern. Diese Dinge sind sehr komplex. Die Technik der Add-ins reicht dafür nicht mehr aus. Das VSIP stellt Tools und Informationen bereit, die benötigt werden, um solche Produkte in Visual Studio .NET zu integrieren. Es ist aber auch eine Plattform, die VSIP Partnern die Möglichkeit gibt, detaillierte Informationen über die Entwicklungsumgebung in Erfahrung zu bringen.

## 2.4.Evaluierung und Auswahl der geeigneten Technik

Wie schon in Abschnitt 2.1 beschrieben, bieten VSMacros die Möglichkeit, Benutzeraktionen zu automatisieren. Man kann jedoch keine eigenen Tool Windows erzeugen, was ein starkes Argument gegen VSMacros ist. Es bietet sich nämlich an, die vorhandenen Jobs in einem eigenem Tool Window anzuzeigen. Ebenfalls scheint die Erstellung von eigenen Dialogboxen nicht möglich zu sein, was jedoch für die Verwaltung von Benutzerdaten wie Name und Passwort erforderlich wäre.
VSMacros stellen also für unsere Zwecke zu wenig Funktionalität bereit.

Mit Add-ins könnte man fast alles entwickeln, was unseren Anforderungen genügt. Es ist jedoch nicht möglich eigene neue Projekttypen zu erstellen. Es war jedoch angedacht, für ein DCL Experiment einen neuen Projekttyp anzulegen. Daraus folgt, dass das DCL-Frontend mit Hilfe des in Abschnitt 2.3 vorgestellten Visual Studio Integrator Programs entwickelt werden müsste.

Bei der Einarbeitung in die Entwicklung für einen neuen Projekttyp mit Hilfe des VSIP stellten wir schnell fest, dass dies einige fortgeschrittene Kenntnisse in COM, C++ und IDL (Interface Definition Language) erfordert. Hinzu kam, dass die Implementierung sehr umfangreich erschien. Wir entschieden uns für einen Kompromiss, denn die Implementierung eines neuen Projekttyps war in dem gegebenen Zeitrahmen mit 2 Personen nicht zu schaffen.
Ein neuer Projekttyp war aber auch nicht unbedingt notwendig. Man hätte den Vorteil, dass die bei der Benutzung des DCL nicht notwendige Funktionen, wie zum Beispiel der komplette Build- und Debugprozess, dem Benutzer nicht zur Verfügung stehen. Der Fokus wäre nur auf das DCL gelenkt.

Also entschieden wir uns das DCL-Frontend mit Hilfe der Add-ins in Visual Studio .NET 2003 zu integrieren.

# 3. Entwicklungsprozess

Nachdem die Entscheidung aufgrund von Anforderungen in der Aufgabenstellung für eine der drei Methoden gefallen war, soll dieses Kapitel nun den folgenden Entwicklungsprozess dokumentieren.

Das Ziel der Analysephase war es, die Funktionalität des DCL's möglichst genau zu analysieren, um eben diese auch in dem zu entwickelnden Add-in bereitzustellen. Hierbei war vor allem das bestehende Webinterface zu betrachten, wie auch die zugrunde liegende Webservice-Schnittstelle. Daran gekoppelt waren aber auch softwareergonomische Fragen, wie zum Beispiel die entsprechenden Bedienelemente.
Da ein Großteil des Entwicklungsaufwand sich mit der Kommunikation mit den schon vorhandenen Schnittstellen beschäftigte, erschien es uns nötig, uns zuerst damit vertraut zu machen. Die zwei großen Schnittstellen, mit denen das zu entwickelnde Add-in kommunizieren muss, ist auf der einen Seite die des Visual Studio Automation Model und zum anderen die des Webservices auf der Seite des DCL's. Die Einarbeitung erfolgte wie in Abschnitt 3.2 näher beschrieben und begründet mit Hilfe von Prototypen, von denen bei Kernfunktionalitäten auch Auszüge aus dem Quellcode zum besseren Verständnis dargestellt sind.
Der anschließende Abschnitt Designentscheidungen erklärt die Gesamtstruktur des Add-ins mit Hilfe eines Klassendiagramms. Außerdem wird auf wichtige Entscheidungen eingegangen bezüglich der Nebenläufigkeit, der zu speichernden Daten und der Fehlerbehandlung.
Die Hauptaspekte der Implementierung sind im Abschnitt 3.4 beschrieben, wobei auf ausgewählte Problemstellungen eingegangen wird, sofern sie nicht bei den Prototypen schon beschrieben wurden.
Im letzten Unterabschnitt geht es um die Performance. Aufgrund der anfänglichen Unzufriedenheit mit der Geschwindigkeit der Kommunikation zwischen Add-in und dem Webservice des DCL's, wurden in einem separaten Tool Messungen bei unterschiedlichsten Verbindungsarten vorgenommen.

## 3.1. Analyse

Bei einer Softwareentwicklung ist im Allgemeinen das Ziel der Analysephase die Zusammenfassung aller fachlichen Anforderungen an die Software. Dies lässt sich doch in unserem Fall dahingehend vereinfachen, dass im optimalen Fall das Add-in sämtliche Funktionalität des DCL's, die auch im Webinterface zur Verfügung steht, in integrierter Form in Visual Studio .NET 2003 anbietet.
Daher kommt man relativ schnell zu der folgenden Auflistung, die praktisch eine Art verkürztes Pflichtenheft darstellt:

- Einloggen: Der Benutzer muss die Möglichkeit haben wie im Webinterface seinen Loginnamen und sein Passwort einzugeben. Zusätzlich sollte das Add-in auch ermöglichen, die Daten persistent zu speichern.
  In der Benutzeroberfläche bedeutet dies, dass es ein Dialogfeld gibt, in dem die Daten einmalig eingegeben werden. Dies sollte in einem Menü aufgerufen werden können.

- Benutzen von Experimenten: Die Eingabe von Code, der von einem bestimmten Experiment ausgeführt werden soll, ist eine der Hauptfunktionen des DCLs.
  Aus ergonomischer Sicht haben wir uns hier aufgrund der Wichtigkeit dieser Funktion für zwei Möglichkeiten entschieden. Zum einem sollte für jedes Experiment ein Eintrag im Menü existieren, zum anderen sollte es aber auch eine Art Auswahlbox in der Symbolleiste geben, um eventuell diese Funktion schneller zugänglich zu machen. Eine weitere Überlegung war die Verwendung eines Untermenüs für das Senden eines Jobs. Da die Funktion oft benutzt wird und die Zahl der Experimente zehn bis zwölf nicht übersteigt, haben wir uns dagegen entschieden.
- Verwalten von Jobs: Hierzu gibt es folgende Grundfunktionen die angeboten werden müssen: Zum einen sollten in irgendeiner Weise alle dem Benutzer zugeordneten Jobs mit dem aktuellen Status angezeigt werden. Zusätzlich sollten Jobs abgebrochen bzw. gelöscht werden können.
  Um dies in Visual Studio .NET zu integrieren, käme ein Tool Window in Frage, in dem angelehnt an den Solution Explorer die Experimente und Jobs in einer Baumstruktur angezeigt werden. Für die Funktionen Abbrechen und Löschen empfiehlt sich dann ein Kontextmenü.
- Anzeigen der Ergebnisse: Das DCL generiert für jeden Job eine HTML Seite, auf der die Ergebnisse und der Code des Jobs angezeigt werden.
  Zum Aufrufen der Funktionen käme ebenfalls das Kontextmenü in Frage. Die Ergebnisseite kann von Visual Studio angezeigt werden. Als spezielles Ergebnis kann der Compileroutput auch wie in Visual Studio üblich im Output Window angezeigt werden. Beim Anzeigen des Codes sollte ein separates Textfenster in Visual Studio geöffnet werden, um ihn eventuell lokal speichern zu können.
- Anzeigen der Kameras: Um das Experiment zu überwachen bietet das DCL Kameras.
  Da die Kameras nicht durch eine Webservice-Schnittstelle einem Experiment zugeordnet werden können, kann die Kameraansicht nicht im oben genannten Kontextmenü aufgerufen werden. Alternativ bietet sich das Menü an. Da die Kameras nicht so wichtig sind wie das Senden, würde hier ein Untermenüpunkt ausreichen, in dem es für jede Kamera einen Eintrag gibt. Die Anzeige der Kameras kann wie im Webinterface in einem neuen Fenster angepasster Größe erfolgen.

Die aufgelisteten Funktionen und ihre Umsetzung in Bedienelemente in Visual Studio .NET, soll im folgenden Kapitel auf ihre Machbarkeit unter Benutzung des Automation Models überprüft werden. Das in der Analysephase entwickelte Modell sollte vollständig und konsistent sein. Da wir Funktionen der Webservice-Schnittstelle mit einbezogen haben, ist diese Eigenschaft gegeben. Die Realisierbarkeit wird im nächsten Abschnitt untersucht und streng genommen ist erst dann die Analyse abgeschlossen.

## 3.2. Prototypen einzelner Funktionen

Aufgrund der teilweise unvollständigen Dokumentation des Automation Models von Visual Studio .NET 2003 haben wir uns entschieden, bei der Einarbeitung in die Schnittstellen kleine Add-ins zu verwenden. Ziel dieser Beispielanwendungen, die jeweils nur einen kleinen Teil der Funktionalität exemplarisch implementieren war es, die Machbarkeit an Stellen, die uns kritisch erschienen zu überprüfen. Dies ermöglichte uns die nachfolgenden Designentscheidungen auf einer fundierten Grundlagen zu fällen.
Durch den Wizard zur Erstellung von Add-ins ist es relativ einfach, ein lauffähiges Add-in zu erzeugen, welches das Interface Extensibility.IDTExtensibility2 implementiert. Jedes Add-in besitzt damit auch neben einigen anderen Methoden die Methode OnConnection(...), in dem es ein DTE-Objekt übergeben bekommt (siehe Code-Beispiel 1). Durch dieses Objekt hat das Add-in Zugriff auf alle Erweiterungsmöglichkeiten der IDE. Da der Quellcode dieser

Prototypen zu großen Teilen später modifiziert übernommen werden konnte, war der zusätzliche Aufwand vergleichsweise gering.

```
namespace MyAddin1
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;
    // The object for implementing an Add-in.
    [GuidAttribute("8FCD12C2-8A4C-42DB-A21C-5C0DF9A60A25"),
        ProgId("MyAddin1.Connect")]
    public class Connect : Object, Extensibility.IDTExtensibility2
    {
        public Connect() {}
        public void OnConnection(object application,
                Extensibility.ext_ConnectMode connectMode,
                object addInInst,
                ref System.Array custom)
        {
            applicationObject = (_DTE)application;
            addInInstance = (AddIn)addInInst;
        }
        public void OnDisconnection(Extensibility.ext_DisconnectMode
                disconnectMode,
                ref System.Array custom) {}
        public void OnAddInsUpdate(ref System.Array custom) {}
        public void OnStartupComplete(ref System.Array custom) {}
        public void OnBeginShutdown(ref System.Array custom) {}

        private _DTE applicationObject;
        private AddIn addInInstance;
    }
}
```

**Code-Beispiel 1: Einfachstes lauffähiges Add-in, das durch den Wizard erzeugt wurde**

Nach dem „Extreme Programming“ Ansatz wurde für jeden Prototypen eine kleine, überschaubare und eindeutig überprüfbare Aufgabenstellung gewählt. Im Folgenden sind die Anforderungen an die einzelnen Prototypen aufgezählt und die Probleme bzw. Erkenntnisse bei deren Realisierung.

### 3.2.1. Prototyp 1 – Einfügen eines Menüs und einer Toolbar in die IDE

Zweck:
Die Funktionen des Add-ins müssen aufgerufen werden können. Dies setzt man mit Hilfe von Menüeinträgen oder einer Toolbar um.

Anforderung:

- Einfügen eines eigenen Menüs im Hauptmenü
- Hinzufügen von Menüeinträgen und Untermenüs
- Erstellen einer Toolbar

Realisierung:
In Visual Studio sind alle Menüs und Toolbars so genannte CommandBars. Das DTE-Objekt hält eine Referenz auf die Collection. So kann man leicht auf eine CommandBar zugreifen, sofern man den Namen kennt. Jede CommandBar hat eine Collection von CommandBarControls, wobei ein CommandBarControl entweder vom Typ CommandBarButton, CommandBarComboBox oder CommandBarPopup sein kann.

Mit Hilfe des CommandBarCombobox-Objekt erzeugt bzw. manipuliert man eine EditBox, eine DropdownList oder eine ComboBox auf der Toolbar. Mit dem CommandBarPopup-Objekt werden Untermenüs repräsentiert. Das CommandBarButton-Objekt steht sowohl für normale Kommandos auf der Toolbar als auch im Menü. Zusätzlich zum CommandBarButton-Objekt gibt es noch das Command-Objekt. Es stellt das eigentliche Kommando dar.
Um dem Hauptmenü nun einen neuen Eintrag hinzuzufügen, benötigt man als erstes die Referenz auf das dazugehörige CommandBar-Object. Wie in Code-Beispiel 2 zu sehen ist, erhält man die Referenz über Namen der CommandBar mit Hilfe der CommandBar-Collection. Ein neuer Eintrag kann nun mit der Methode Add(…) hinzugefügt werden.

```
// Create 'DCL' main menu item
CommandBar MainMenuBar = (CommandBar)applicationObject.CommandBars["MenuBar"];
CommandBarControl dclMenuBarControl;
dclMenuBarControl = MainMenuBar.Controls.Add(
                        MsoControlType.msoControlPopup,
                        Type.Missing,
                        Type.Missing,
                        1,
                        true);
dclMenuBarControl.Caption = "&DCL";
dclMenuBarControl.Visible = true;
```

**Code-Beispiel 2: Erzeugen ein Eintrags im Hauptmenü**

Jedes Add-in, das eigene Kommandos erzeugt, muss die IDTCommandTarget Schnittstelle implementieren und damit die Methoden Querystatus(…) und Exec(..). Die Methode Exec(…) wird ausgeführt, nachdem der Benutzer ein Kommando aufgerufen hat. Als Argument bekommt man unter anderem den Namen des Kommandos, das ausgeführt werden soll. Deshalb ist bei der Erzeugung eines Command-Objekts der Name besonders wichtig. Das Erzeugen eines Kommandos erfolgt mit der Methode AddNamedCommand(…), die ein Command-Objekt zurück liefert, wie das folgende Code-Beispiel 3 zeigt.
Das Kommando existiert zwar, ist jedoch noch keiner CommandBar zugeordnet und kann somit noch nicht aufgerufen werden. Mit der Methode AddControl(…) kann man ein Kommando beliebig vielen CommandBars hinzufügen.
Um herauszufinden, wie man Menüeinträge voneinander mit Hilfe eines Separators abgrenzt, war eine längere Suche nötig. Es war schließlich das Attribut BeginGroup, das auf true gesetzt werden musste.

```
// Create new menu item (command)
object []contextGUIDS = new object[] {};
Command cmd = null;
cmd = applicationObject.Commands.AddNamedCommand(addInInstance,
                            name,
                            caption,
                            toolTip,
                            true,
                            iconFaceID,
                            ref contextGUIDS,
                            (int)vsCommandStatus.vsCommandStatusSupported +
                            (int)vsCommandStatus.vsCommandStatusEnabled);
// Assign command to menu (command bar)
CommandBarControl cmdBarControl = cmd.AddControl(cmdBar, 1);
// Create menu item separator
cmdBarControl.BeginGroup = true;
```

**Code-Beispiel 3: Erzeugen eines Kommandos**

Die Erstellung einer Toolbar realisierten wir mit der Methode AddNamedCommandBar(…). Der Rückgabewert ist eine CommandBar, der beliebig viele CommandBarControl-Objekte hinzugefügt werden können.

Ein kleines Problem bei der Erstellung der Kommandos und der Toolbar war die Persistenz. Prinzipiell gibt es ein Argument der Methode Add(…) mit dem man einstellen kann, ob die Kommandos bzw. die Toolbars nach dem Schließen von Visual Studio .NET erhalten bleiben oder nicht. Die vom Automation Model dokumentierten Methoden zur Erzeugung von Command- und CommandBar-Objekten sind aber AddNamedCommand(…) bzw. AddNamedCommandBar(…). Diese bewirken, dass Kommandos und Menüs nach dem Schließen persistent bleiben. Weiterhin stand in der Dokumentation, dass nach dem Deinstallieren des Add-in's, die Kommandos bzw. Toolbars wieder gelöscht werden. Wir stellten jedoch fest, das dies bei den Toolbars nicht der Fall war.
Eine andere Alternative wäre die Benutzung der Methode Add(…) gewesen. Man hätte bei der Erzeugung festgelegt, dass Toolbars nicht persistent bleiben. Dies hat jedoch zur Folge, dass Toolbars jedes Mal beim Start neu platziert werden müssten, was sehr unkomfortabel für den Benutzer ist. So entschieden wir uns sie persistent zu erzeugen. Die Lösung für die Löschung von Toolbars nach der Deinstallation steht noch aus.

### 3.2.2. Prototyp 2 – Erzeugen eines Tool Windows

Zweck:
Um dem Benutzer eine Verwaltung seiner Jobs zu ermöglichen, soll ein Tool Window erzeugt werden, in dem die verfügbaren Experimente und deren Jobs angezeigt werden können. Mit Hilfe eines Kontextmenüs sollen für Jobs Aktionen ausgeführt werden können.

Anforderung:
- Erzeugen und Anzeigen eines Tool Windows
- Beispielhafte Erstellung einer TreeView
- Erzeugung eines Kontextmenüs

Realisierung:
Alle Fenster (Windows) werden durch das Window-Objekt repräsentiert. Die Erzeugung eines neuen eigenen Tool Windows ist in Visual Studio .NET 2003 etwas umständlich. Tool Windows werden mit der Methode CreateToolWindow(…) erstellt und können nur ActiveX-Controls darstellen, die jedoch mit .NET nicht erstellt werden können. Eine Realisierung mit ATL erschien uns zu aufwendig, da wir so gut wie keine Erfahrung damit haben. Deshalb schauten wir uns nach einer Alternative um, die schnell gefunden war. Die Lösung ist ein ActiveX-Control, das ein .NET UserControl lädt. Den Code entnahmen wir komplett aus einem Beispiel. Doch dieses Beispiel war für unseren Einsatz nicht ganz vollständig. Das ActiveX-Control hatte zwar Methoden zum Laden, bot jedoch keine Möglichkeit an die Referenz des UserControls zu gelangen.
Nach weiteren Recherchen fanden wir heraus, dass es im Namespace System.Runtime.Remoting eine Klasse ObjectHandle gibt, mit der man Objektinstanzen zwischen verschiedenen Application Domains austauschen kann. Dem ActiveX-Control musste deshalb die Methode GetObjectHandle() hinzugefügt werden, die in Code-Beispiel 4 zu sehen ist.

```
HRESULT CVSUserControlHostCtl::GetObjectHandle ( IUnknown ** ppObjHandle )
{
    return (m_pObjHandle->QueryInterface (IID_IObjectHandle,
                                          (LPVOID*)ppObjHandle));
}
```

**Code-Beispiel 4: Zum ActiveX-Control hinzugefügte Methode**

Mit dieser Lösung hatten wir die Möglichkeit, aus unserem Add-in heraus ein .NET UserControl zu laden und weitere Methoden dieses Objektes aufzurufen.

Das folgende Code-Beispiel 5 zeigt den kompletten Code zum Erzeugen des Tool Windows, das ein .NET UserControl anzeigt. Die Klasse JobExplorer implementiert in diesem Fall ein System.Windows.Forms.UserControl.

```
object objTemp = null;
jobExplorerWindow = applicationObject.Windows.CreateToolWindow (addInInstance,
                        "VSUserControlHost.VSUserControlHostCtlDCL",
                        "Job Explorer",
                        jobExpWinCLSID,
                        ref objTemp);

// When using the hosting control, you must set visible to true before calling
// HostUserControl, otherwise the UserControl cannot be hosted properly.
jobExplorerWindow.Visible = true;

Assembly asm = Assembly.GetExecutingAssembly();

IVSUserControlHostCtl objControl;
objControl = (IVSUserControlHostCtl)objTemp;
// Loads the JobExplorer(UserControl) in VSUserControlhost
objControl.HostUserControl(asm.Location, "DCL_VSI.JobExplorer");
// Ask the VSUserControlHost for the ObjectHandle of the hosted control
ObjectHandle jobExpObjHandle = (ObjectHandle) objControl.GetObjectHandle();
// Unwrap the object handle to get the 'JobExplorer object' hosted in
// VSUserControl
jobExp = (JobExplorer) jobExpObjHandle.Unwrap();
```

**Code-Beispiel 5: Erzeugen eines Tool Windows**

Um ActiveX-Controls benutzen zu können, muss man sie in der Registry-Datenbank registrieren. Nach längerer Suche fanden wir schließlich heraus, dass es im Deployment Project die Eigenschaft Register gibt. Nun muss man nur noch für die entsprechende Dll den Wert auf vsdrfCOMSelfReg setzen, da dieses ActiveX-Control bereits Methoden für Registrierung enthält.
Zusätzlich mussten wir beachten, dass wir dieses ActiveX-Control modifiziert hatten. Das zog einige Änderungen nach sich. Sämtliche GUIDs und die ProgID mussten ausgetauscht werden, da es sonst Versionskonflikte mit dem ursprünglichen ActiveX-Control hätte geben können.

Da wir mit dieser Lösung in der Lage waren, ein UserControl anzuzeigen, stellte die Erzeugung einer TreeView kein Problem dar. Interessanter wurde es beim Kontextmenu. Zuerst verwendeten wir das Kontextmenu von WinForms. Doch dieses Menü besaß ein anderes Look and Feel als das des Visual Studio's, wie die folgende Abbildung 2 zeigt.

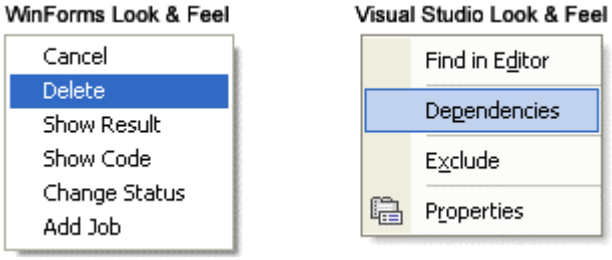


**Abbildung 2: Vergleich des Look & Feels der Kontextmenüs**

Wir fanden eine GUI Bibliothek namens DotNetMagic, die ein Kontextmenu im Style von Visual Studio .NET 2003 zur Verfügung stellte. Diese Bibliothek ist jetzt kommerziell. Wir benutzten jedoch eine Vorgängerversion, die als Freeware zu haben ist.

### 3.2.3. Prototyp 3 – Inhalt eines Dokuments und das Output Window

Zweck:
Wenn der Benutzer Code an das DCL senden will, muss das Add-in das entsprechende Dokument herausfinden und den Inhalt einlesen. Anschließend sollen Compilermeldungen des DCL ausgegeben werden.

Anforderungen:

- Herausfinden des aktiven Dokuments
- Einlesen des Inhalts
- Ein Meldung im Output Window ausgeben

Realisierung:
Dokumente werden durch Document-Objects repräsentiert. Das DTE-Object hält eine Referenz auf das aktive Dokument. Mit der Eigenschaft Fullname kann man den Dateinamen herausfinden und somit den Inhalt einlesen. Es kann vorkommen, dass der aktuelle Inhalt, nicht mit dem Inhalt der Datei übereinstimmt. Das ist dann der Fall, wenn das Dokument editiert wurde und die Änderungen noch nicht gespeichert wurden. Deshalb sollte man zuerst das Dokument mit der Methode Save(…) speichern, bevor der Inhalt ausgelesen werden kann.
Um eine Meldung im Output Window ausgeben zu können, benötigt man zuerst ein spezielles Window-Object, das OutputWindow-Object. Grundsätzlich erhält man ein Window-Object über die Windows-Collection des DTE-Objects. Mit der Angabe vorgegebener Konstanten bestimmt man, für welches Window man eine Referenz haben möchte. In diesem Fall ist der Wert Constants.vsWindowKindOutput. Doch die Window-Collection liefert nur das´Window-Object und nicht das OutputWindow-Objekt. Dieses versteckt sich in der Eigenschaft Object und muss in ein OutputWindow umgewandelt werden. Um letztendlich eine Meldung ausgeben zu können benötigt man ein OutputWindowPane-Object. Entweder man bedient sich eines Vorhandenen aus der dazugehörigen Collection oder erzeugt ein Neues. Dann kann die Methode OutputString(…) mit der gewünschten Meldung als Argument aufgerufen werden.
Das folgende Code-Beispiel 6 zeigt den eben beschriebenen Algorithmus.

```
// Get the OutputWindow
win = applicationObject.Windows.Item(Constants.vsWindowKindOutput);
outputWindow = (OutputWindow) win.Object;
}
// Find or create OutputPane
outputPane = outputWindow.OutputWindowPanes.Add ("DCL - Compiler Output");
// Write text to OutputPane
outputPane.OutputString(text);
```

**Code-Beispiel 6: Ausgabe im Output Window**

### 3.2.4. Prototyp 4 – Anzeigen des Codes und der Ergebnisse eines Jobs

Zweck:
Zu einem beliebigen Job soll sich der Benutzer den ausgeführten Code vom DCL herunterladen können, der automatisch in einem neuen Textdokument angezeigt wird. Für weitere Ergebnisse des Jobs besteht die Möglichkeit, die Webseite in dem in Visual Studio integrierten Webbrowser zu laden.

Anforderungen:

- Erzeugung eines neuen Textdokuments
- Schreiben von Text in dieses Dokument
- Anzeigen einer Url im integrierten Webbrowser

Realisierung:
Um ein neues Textdokument zu erzeugen, greift man auf die Funktion NewFile(…) zurück. Das DTE-Object hält dafür eine Collection von ItemOperations bereit. Der Rückgabewert der Methode NewFile(…) ist ein Window-Object. Das Document-Attribut liefert eine Referenz auf das neu erzeugte Dokument. Wie schon in Abschnitt 3.2.3 erläutert, wird ein Dokument intern durch das Document-Object repräsentiert. Wir benötigen jedoch ein TextDocument-Object, um Schreiboperationen durchführen zu können. Die Methode Object(…) gibt das spezielle Document-Object zurück. Um nun Text einfügen zu können, muss zunächst eine Einfügemarke definiert werden. Dies ist ein EditPoint-Object, das durch die Methode CreateEditPoint(…) erzeugt wird. Um schließlich den Text einzufügen, muss nur noch die Methode Insert(…) des EditPoint-Objects aufgerufen werden.

```
// Create new text document
win = applicationObject.ItemOperations.NewFile("General\\Text File",
                              filename,
                              Constants.vsViewKindCode);
// Get the TextDocument-Object of the new file
doc = win.Document;
doc.Activate();
textDoc = (TextDocument) doc.Object("TextDocument");
// Get an EditPoint-Object in the TextDocument
editPoint = textDoc.CreateEditPoint(textDoc.StartPoint);
// Write to the TextDocument
editPoint.Insert(code);
```

**Code-Beispiel 7: Erzeugen eines neuen Textdokuments**

Die Anzeige einer bestimmten Webseite im integrierten Webbrowser erschien uns auf den ersten Blick nicht möglich zu sein. Man erhält zwar Zugriff auf das Window-Object des Webbrowsers (analog zum Output Window; siehe 3.2.3), jedoch gibt es im Automation Model kein dokumentiertes spezielles Object, was den Browser repräsentiert. Somit dachten wir, dass die Navigation zu einer bestimmten Url nicht realisierbar sei. Doch dann fanden wir den Artikel von John Robbins [5]. Er schrieb, dass das Attribut Object eine Referenz auf das Webbrowser-Control-Object hat. Mit dieser Information konnten wir die Methode Navigate(…) des Webbrowser-Controls aufrufen und so die gewünschte Webseite im Webbrowser laden. Das folgende Code-Beispiel 8 zeigt den kompletten Ablauf.

```
// Get the browser window.
Window webBrowserWindow =
        applicationObject.Windows.Item(Constants.vsWindowKindWebBrowser);

// Force the window to be visible.
webBrowserWindow.Visible = true ;
// Get the IE control inside the browser window.
SHDocVw.WebBrowser webBrowser = null;
webBrowser = (SHDocVw.WebBrowser)webBrowserWindow.Object;

Object zero = 0 ;
Object emptyString = "";
// Navigate to the Url.
webBrowser.Navigate (url, ref zero, ref emptyString, ref emptyString,
               ref emptyString);
```

**Code-Beispiel 8: Laden einer Webseite im integrierten Webbrowser**

### 3.2.5. Prototyp 5 – Schnittstellen zum DCL via Webservice

Zweck:
Wichtigste Aufgabe des Add-ins ist die Kommunikation mit den Webservices des DCLs. Diese wesentliche Schnittstelle soll in diesem Prototyp getestet werden.

Anforderungen:

- Verwendung des Webservice
- Auflistung der vorhandenen Experimente
- Auflistung der vorhandenen Tickets eines Benutzers

Realisierung:
Nach dem Hinzufügen einer Webreference erzeugt Visual Studio automatisch Klassen, die den Zugriff auf die Webservice-Schnittstellen realisieren. Nun können die in [6] beschriebenen Methoden leicht benutzt werden. Das Code-Beispiel 9 zeigt die exemplarische Verwendung der Methoden ListExperiments() und GetTicketList(…).

```
de.discourse.was;
de.discourse.was1;

public void connect ()
{
    DCLTicketServer TS = new DCLTicketServer();
    DCLExperimentService ES = new DCLExperimentService();

    string[] experiments = null;
    experiments = ES.ListExperiments();

    string[] ticketList = null;
    bool viewAll = true;
    ticlist = TS.GetTicketList("testaccount", "123456", out viewAll);
}
```

**Code-Beispiel 9: Benutzung der Webservice-Schnittstellen**

## 3.3. Designentscheidungen

In diesem Abschnitt soll zuerst der allgemeine Aufbau von DCL Visual Studio Integration erläutert werden und anschließend spezieller auf die Nebenläufigkeit, die Einstellungen und die Fehlerbehandlung eingegangen werden.
Die Struktur des Add-ins ist in Abbildung 3 in Form eines Klassendiagramms dargestellt. Wesentlich sind hierbei die drei Klassen DCL_VSI, JobExplorer und Settings, die fast die ganze Funktionalität des Add-ins realisieren (siehe Abbildung 3).
In DCL_VSI sind die Methoden, die die Kommunikation mit der IDE umsetzen. Ein Objekt der Klasse DCL_VSI wird beim Start in der Klasse Connect erzeugt. Danach initialisiert die Methode startup() das Add-in. Hier werden die Menüs erzeugt und gelöscht, der Job Explorer erzeugt und eine Nachricht an ein Objekt der Klasse Settings gesendet, die Einstellungen zu lesen (siehe Abschnitt 3.3.2). Nach der Initialisierung existiert von jedem der drei Hauptklassen genau ein Objekt.

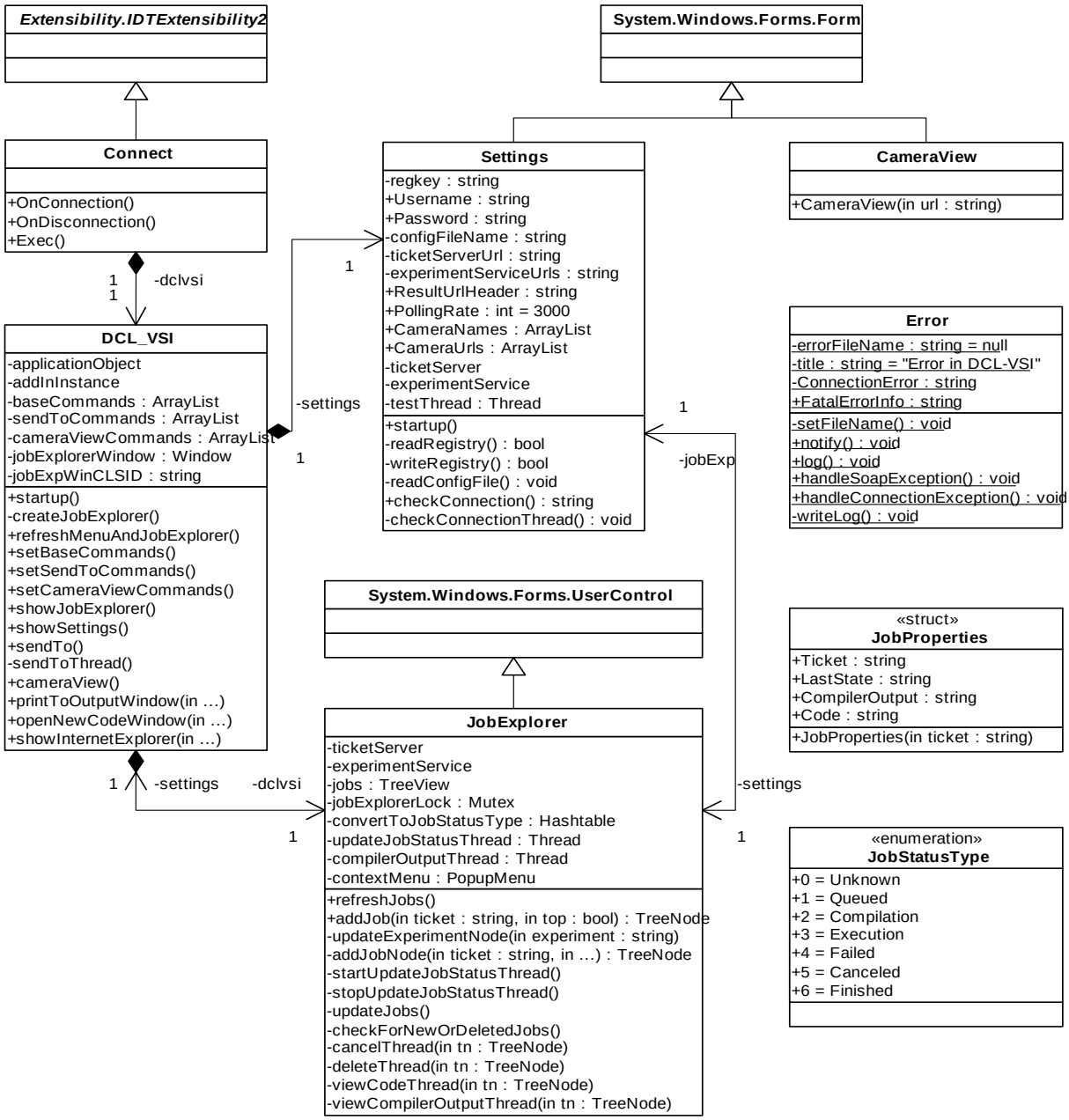


**Abbildung 3: Klassendiagramm des DCL VSI**

Die Klasse JobExplorer realisiert die Kommunikation mit dem DCL. Sie übernimmt ebenfalls die Anzeige der vorhandenen Jobs in einer TreeView. Diese TreeView verwenden wir auch gleichzeitig als Datenstruktur zur Speicherung der Informationen über einen Job. Hierzu hat jeder Knoten in der TreeView, der einen Job repräsentiert, als „Tag“ eine JobProperties-Struktur. Diese Struktur beinhaltet den Ticketstring, den letzten abgefragten Status, und wenn einmal aus dem DCL abgefragt, den Compileroutput und den Code. Grund für das lokale Speichern ist die Tatsache, dass sich Compileroutput und Code, wurden sie einmal aus dem DCL gelesen, sich nicht mehr ändern. Fragt der Nutzer diese Werte nun zum Beispiel ein zweites Mal ab, muss keine weitere Abfrage erfolgen.
Die TreeView jobs beinhaltet also alle Daten zu den Jobs, deren Status aber in regelmäßigen Abständen aktualisiert werden müssen. Dies geschieht in einem separaten Thread wie in Abschnitt 3.3.1 näher erklärt.
Ein weiterer erwähnenswerter Punkt ist die Konsistenz der TreeView. So wird zum Beispiel beim Abbrechen eines Jobs nicht der Status in der TreeView direkt manipuliert, sondern der entsprechende TreeNode wird anschließend aktualisiert. Somit ist sichergestellt, dass die TreeView zu jedem Zeitpunkt dem Zustand des DCLs entspricht. Zusätzlich zu beachten ist, dass Jobs auch im Webinterface hinzugefügt und gelöscht werden können. Um trotzdem Konsistenz zum DCL zu erreichen, wird in regelmäßigen Abständen, jedoch nicht so oft wie die Aktualisierungen der laufenden Jobs, die Methode checkForNewOrDeletedJobs() ausgeführt, um alle vorhandenen Tickets zu überprüfen.

### 3.3.1. Nebenläufigkeit

Da die Kommunikation mit dem DCL je nach Verbindungsart sehr lange dauern kann (siehe Abschnitt 3.5) und der Code des Add-ins im Thread der IDE läuft, ist für diese Zeit keine Interaktion mit der gesamte IDE möglich. Dieser Sachverhalt macht deutlich, dass mehrere Threads von Nöten sind, um eine komfortable Bedienung möglich zu machen. Es galt deshalb der Grundsatz während der ganzen Entwicklung, dass sobald mit dem Webservice interagiert wird, dieser Codeteil in einem separaten Thread laufen muss, damit die IDE fähig ist, weiter auf den Benutzer zu reagieren.

Dabei gibt es aber zwei verschiedene Arten von Threads. Zum einen die, die einmalig gestartet werden, ihre zugewiesene Methode abarbeiten und damit beendet sind. Dies ist zum Beispiel das Senden eines Jobs, das Abbrechen oder das Anzeigen des Codes der Fall. Sendet der Benutzer kurz hintereinander mehrere Jobs, so können beliebig viele Threads, die jeweils einen Job senden, gleichzeitig abgewickelt werden. Zum anderen gibt es aber auch solche Threads, die eine bestimmte Aufgabe haben, und deshalb nur einmal existieren. Die Aktualisierung des Job Explorers zum Beispiel wird nur von einem Thread ausgeführt. Es wird jedoch keine zwei Threads geben, die mit dieser Aufgabe betraut sind. Ähnlich steht der Sachverhalt beim Test einer Verbindung im Settingsdialog und bei der Ausgabe des aktuellen Zustands eines Jobs im Output Window. Möchte der Benutzer einen anderen Job im Output Window angezeigt haben, so wird eben der vorhandene Thread zuerst abgebrochen.

Dadurch dass theoretisch beliebig viele Threads gleichzeitig laufen können, ist ein besonderes Augenmerk auf die Synchronisation zu legen. Hier spielt das Mutex „jobExplorerLock“ eine zentrale Rolle. Es dient dazu sicherzustellen, dass immer nur ein Thread gleichzeitig Zugriff auf den Job Explorer bzw. auf die TreeView hat. Dadurch wird zwar ein Großteil der Codeausführung wieder sequenziell abgearbeitet, jedoch hat dies keine Auswirkung auf die Performance, da die Antworten des DCL-Servers den Hauptteil der Zeit in Anspruch nehmen.

### 3.3.2.Einstellungen

Bei den Einstellungen gibt es zwei Typen von Daten, die persistent gespeichert werden müssen. Zum einen sind dies die benutzerspezifischen Daten wie Loginname und Passwort. Diese Daten sollten vor unbefugtem Zugriff gesichert sein. Hierbei haben wir die Tatsache vorausgesetzt, dass alle Einträge in der Registrierungsdatenbank unter „HKEY_CURRENT_USER“ für andere User unzugänglich sind. Die Installationsroutine legt die entsprechenden Schlüssel an und beim ersten Starten werden die eingegebenen Daten dort gespeichert. Eine Verschlüsselung der Passwörter mittels Hashfunktionen wie MD5 wird vom DCL zur Zeit nicht unterstützt.
Der zweite Typ Daten sind eher administrative Einstellungen. Sie müssen nicht vor unbefugtem Zugriff geschützt werden, sondern können von allen eingesehen und editiert werden. Zu diesem Typ Einstellungen gehören diverse Urls des DCLs, die Aktualisierungsrate des Job Explorers und die Urls der Kameras. Wir haben uns entschieden diese Daten in einer XML-Datei zu speichern, die beim Starten von Visual Studio .NET einmalig gelesen wird. Das genaue Format dieser Datei ist in Abschnitt 4.1 näher erläutert.

Das Lesen und Schreiben in der Registrierungsdatenbank, das Lesen der XML-Datei und der Dialog zur Eingabe der Benutzerdaten sollen in der Klasse Settings realisiert werden. Ebenfalls in der Klasse Settings ist die Methode zum Testen der Verbindung.

### 3.3.3. Fehlerbehandlung

Zur einheitlichen und zentralen Fehlerbehandlung dient die Klasse Error. Hier werden Fehlermeldungen für den Benutzer generiert. Außerdem werden Exceptions in eine Logdatei geschrieben. Beim Auftreten einer SoapException und eines Verbindungsabbruches werden besondere Methoden von der Klasse Error bereitgestellt.

## 3.4. Implementierung

Das ganze Add-in wurde mit der Programmiersprache C# entwickelt. Die Ausnahme stellt das ActiveX-Control dar, das unter der Benutzung von ATL in C++ geschrieben wurde.
Die von uns verwendeten Icons sind entweder selbst erstellt, oder wurden von Microsoft Office übernommen.

Im Folgenden sollen ausgewählte Aspekte bestimmter Implementierungen vorgestellt werden.

### 3.4.1. Senden

Der Auftrag für das Senden von Jobs erfolgt wahlweise über das Menü oder die Toolbar. Nachdem der Benutzer die Sendefunktion ausgelöst hat, muss zunächst das Dokument ermittelt werden, das den Code enthält. Wie schon in 3.2.3 beschrieben, ist dies über die Eigenschaft ActiveDocument des DTE-Objects möglich. Doch es kann das Problem auftreten, dass das ActiveDocument nicht sichtbar ist. Das liegt daran, dass dem Benutzer gerade der Webbrowser oder die Hilfe angezeigt wird. In diesem Fall verweist das Attribut ActiveDocument auf das zuletzt bearbeitete Dokument. Für den Benutzer wäre so nicht ersichtlich, welches Dokument gesendet wurde. Leider gibt es im Automation Model keine Möglichkeit wirklich zu prüfen, ob ein Fenster gerade im Vordergrund ist oder nicht. Ist also zum Beispiel im Hauptfenster der Webbrowser, das Hilfefenster und ein Dokument geöffnet, so ist bei allen drei die Eigenschaft Visible auf true gesetzt. Doch nur ein Fenster ist in diesem Moment vollständig zu sehen. Aus diesem Grund legten wir fest, dass das Fenster des aktiven Dokuments gleichzeitig das aktive Fenster sein muss. Doch auch bei dieser

Variante kann aus Benutzersicht ein merkwürdiges Verhalten auftreten. Das zu sendende Dokument ist sichtbar und dennoch erscheint eine Fehlermeldung. Das liegt daran, dass ein anderes Fenster gerade den Fokus hat, wie zum Bespiel unser Job Explorer oder vielleicht das Output Fenster.
Trotz dieses Verhaltens entschieden wir uns für die letzte Variante. Der Benutzer wird bei Auftreten dieses Falles darauf aufmerksam gemacht, das gerade ein Tool Window aktiv ist. Zusätzlich weisen wir explizit im Benutzerhandbuch darauf hin (siehe Abschnitt 4.2).

### 3.4.2. Funktionen des Kontextmenüs

Der Job Explorer ist das Kernstück unseres Add-ins. Mit ihm ist der Benutzer in der Lage, alle verfügbaren Funktionen für einen Job auszuführen. Das setzt voraus, dass ihm die Jobs zuverlässig und konsistent angezeigt werden.
Für die Anzeige eines Jobs sind drei Informationen notwendig. Sein Experimenttyp, seine Startzeit und der aktuelle Status. Ein Job besitzt sechs gültige Statustypen: Queued, Compilation, Execution, Failed, Canceled und Finished. Die Abfrage erfolgt mit der Methode GetExperimentStatus(…). Sie liefert einen String als Rückgabewert. Doch dieser ist für die Zustände Failed und Finished nicht eindeutig definiert. Alle diese Rückgabewerte beginnen mit Finished oder Finish, sowohl klein als auch groß geschrieben. Hier mal drei Beispiele: „Finish_Experiment crashed“, „finished_Stop Call Failed“, „finished_StopSignal“. Diese Unterschiede zeigen, dass es selbst mit Parsen nicht möglich ist, den Status eindeutig zu bestimmen. Deshalb implementierten wir eine Hashtable, mit der die uns bekannten Rückgabewerte in eine interne Repräsentation (Enumeration JobStatusType) umgewandelt werden. Unbekannte Werte erhalten den Status Unknown. Der große Nachteil ist die Abhängigkeit vom Experimenttyp. So wie es momentan aussieht, definiert jeder Experimenttyp eigene Statusrückgabewerte für die Zustände Failed und Finished. Dadurch war es uns nicht möglich, das Add-in unabhängig vom Experiment zu implementieren, was aber jedoch ursprünglich unser Ziel war. Unserer Meinung nach, sollte die Methode GetExperimentStatus(…) einen eindeutig definierten Rückgabewert liefern. Es wäre besser, für den momentan mit dem Status kombinierten Result Code (siehe Beispiele) einen neuen Ergebnistyp einzuführen.
Eine weitere Inkonsistenz trat bei dem Status Canceled auf. Befindet sich ein Job gerade in der Ausführung (Execution), kann er durch die Funktion CancelExperiment() abgebrochen werden. Jedoch war anschließend der Status des Jobs nicht Canceled sondern Finished.

Wird ein Job mit der rechten Maustaste ausgewählt, so erscheint ein Kontextmenü. Die Einträge sind abhängig vom Status des Jobs. Damit die IDE weiter auf Eingaben des Benutzers reagieren kann, müssen die ausgewählten Aktionen in einem neuen Thread ausgeführt werden. Dies bedeutet technisch, das ein neues Thread-Objekt erzeugt wird. Der Name der Methode, die beim Start des Threads ausgeführt werden soll, wird als Argument bei der Erzeugung des Delegates ThreadStart übergeben.
Mit dieser Variante können aber keine Argumente an die auszuführende Methode übergeben werden. Dieses Problem könnte man leicht lösen, in dem man eine globale Variable (ein Attribut) einführt, die das aktuell zu übergebende Argument speichert. Dies funktioniert nicht, wenn mehrere Threads unmittelbar nacheinander erzeugt werden können, die alle die gleiche Methode als Startmethode benutzen wollen.
Deshalb erstellten wir die Hilfsklasse CommandArgs(siehe Code-Beispiel 10). Sie enthält das Argument und die Methode(als Delegate), die ausgeführt werden soll. Diese Klasse kann damit für alle Aufrufe verwendet werden, die als Argument einen TreeNode erwarten und den Rückgabetype void haben.

```
private class CommandArgs
{
    private TreeNode treeNode;
    public CommandDelegate Command;

    public CommandArgs(TreeNode tn)
    {
        this.treeNode = tn;
    }
    public void ExecCommand()
    {
        Command(treeNode);
    }
}
```

**Code-Beispiel 10: Die Klasse CommandArgs**

### 3.4.3. Aktualisierung des Jobexplorers

Die angezeigten Jobs werden in regelmäßigen Abständen aktualisiert. Unser Ziel war es, eine sehr effiziente Implementierung zu finden. Natürlich wird sofort klar, dass nur Jobs mit dem Status Queued, Compilation und Execution aktualisiert werden müssen. Eventuell noch das Experiment, da dort die Anzahl der Jobs angezeigt wird, die sich momentan in der Warteschlange befinden.
Entscheidend ist aber das Verfahren um die Jobs zu finden, die den betreffenden Status besitzen. Die erste Variante wäre die Implementierung von Listen, wobei pro Status eine Liste geführt wird. Ein zusätzlicher Verwaltungsaufwand wäre notwendig. Wir entschieden uns für die zweite Variante, bei der es diesen Overhead nicht gibt. Dazu muss man wissen, dass die Jobs in zeitlicher Reihenfolge in der TreeView gespeichert werden. So genügt es, für jedes Experiment die Liste der Jobs, mit dem „jüngsten“ beginnend, von oben nach unten durchzugehen. Der Abbruch erfolgt beim Finden eines Jobs mit dem Status Failed oder Finished. Der Status Canceled ist kein Abbruchkriterium. Hat man zum Beispiel drei Jobs gesendet und sind alle Queued, kann man den zuletzt gesendeten Job abbrechen. Nun beginnt die Liste mit einem Job in dem Status Canceled. Der Algorithmus würde nun aufhören, obwohl noch zwei Jobs aktualisiert werden müssten.

Die Aktualisierung eines Jobs bewirkt, dass die TreeView an der entsprechenden Stelle neu gezeichnet werden muss. Da dies während des Aktualisierungsvorgangs häufiger vorkommt, ist ein leichtes Flackern der TreeView erkennbar. Vermeiden kann man dies mit den Methoden BeginUpdate() und EndUpdate(). Mit ihnen erreicht man, dass Änderungen nicht sofort gezeichnet werden. Doch leider mussten wir feststellen, das die gewünschte Wirkung in unserem Kontext ausblieb. Dies könnte daran liegen, das unser .NET UserControl durch das ActiveX-Control geladen wurde. Eventuell wäre damit das ActiveX-Control für die Neuzeichnung zuständig. Doch auch das ActiveX-Control unterstützt die Funktionalität, die Aktualisierung eines Fensters zu steuern, und zwar mit den Methoden Freeze() und Unfreeze(). Doch die Implementierung und Einarbeitung in ATL erschien uns für das Problem zu aufwendig, da wir nicht wussten, ob wir damit das Problem würden lösen können.

## 3.5. Performancetest

Als wir DCL Visual Studio Integration das erste Mal mit einem Account testeten, der mehrere hunderte Jobs hatte, dauerte das Aufbauen, je nach Verbindung, über eine Minute. Dies nahmen wir zum Anlass, den Code an manchen Stellen zu optimieren und darauf zu achten, möglichst wenig Aufrufe des DCL-Webservice zu benutzen.

Zusätzlich haben wir ein kleines Testtool geschrieben, das die drei Aufrufe, die beim Hinzufügen eines Jobs zur TreeView gebraucht werden, ausführt und dabei die Zeit misst. Code-Beispiel 11 zeigt einen Auszug aus diesem Testtool.

```
for (int i = 0; i < iterations ; i++)
{
    time = Environment.TickCount;

    experimentService.GetExperimentTypeofTicket(ticket);
    experimentService.GetExperimentStatus(ticket);
    ticketServer.GetUsageStart(ticket);

    time = Environment.TickCount - time;

    SW.WriteLine(time.ToString());
}
```

**Code-Beispiel 11: Aufrufe des DCL-Webservice im Testtool**

Wir haben dieses Testtool (siehe Abbildung 4) mit den unterschiedlichsten Verbindungsarten ausgeführt, da das DCL auch die Intention hat, die Experimente über das Internet von überall zugänglich zu machen und nicht nur im universitätsinternen Netzwerk. Die Anzahl der Wiederholungen haben wir bei 100 festgesetzt.

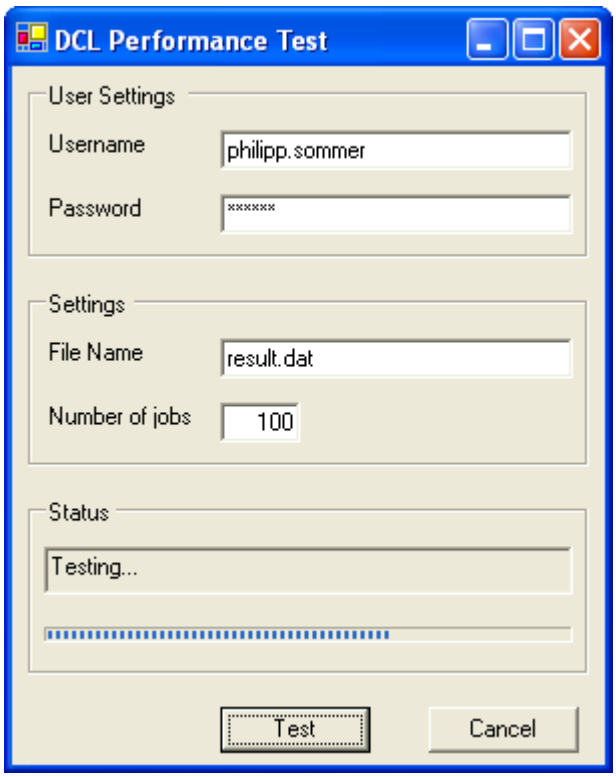


**Abbildung 4: DCL Performance Test**

| Art der Verbindung | durchschnittliche Zeit in ms | minimale Zeit in ms | maximale Zeit in ms |
|---|---|---|---|
| Modem (56kBit) | 1713 | 1562 | 2093 |
| ISDN (64kBit) | 983 | 961 | 1072 |
| DSL (786/256 kBit) | 689 | 660 | 1012 |
| Netzwerk (100 MBit) | 73 | 60 | 561 |

**Tabelle 1: Zeiten für drei Funktionsaufrufe des Webservice**

Das Resultat ist in Tabelle 1 zu sehen. Wie zu erwarten war, ist die Performance stark von der jeweiligen Verbindungsart abhängig.
Diese Zeiten könnten jedoch stark verbessert werden, in dem man die vorhandenen DCL Webservice-Schnittstellen erweitert. Es müsste möglich sein, mit nur einer Funktion, Informationen mehrerer Jobs abzurufen. Dann bräuchte man nicht für jeden Job und jede Anfrage einen neuen Aufruf starten.

# 4. Benutzerhandbuch

Dieses Kapitel soll dazu dienen, das Bedienen von DCL Visual Studio Integration zu erklären, ohne Teile der Programmierung verstehen zu müssen. Ein separates Benutzerhandbuch schien uns aufgrund des relativ geringen Umfangs nicht nötig. Um aber trotzdem ein vollständiges  Benutzerhandbuch zu erstellen, sind eventuell Sachverhalte beschrieben, die in den ersten Teilen dieser Dokumentation bereits erwähnt wurden.

## 4.1. Installation und Einstellungen

Um DCL VSI zu installieren und zu konfigurieren, sollten folgende Schritte ausgeführt werden.

- Zuerst muss die Datei setup.exe gestartet werden. Die Installationsroutine kopiert daraufhin die Programmdateien in das angegebene Verzeichnis.
- Im Explorer kann nun die Datei config.xml im Installationspfad mit einem zum Editieren von XML-Dateien geeigneten Programm geöffnet werden (z. B. Notepad), um die Standardwerte zu kontrollieren und gegebenenfalls zu ändern (siehe Abbildung 5). Das äußerste Datenelement „settings“ in der Datei enthält die folgende Datenelemente, in denen wiederum die entsprechenden Werte angegeben werden können:
  - ticketServerUrl: Url des Webservice des Ticketservers.
  - experimentServerUrl: Url des Webservice des Experimentservers.
  - resultUrlHeader: Anfang der Url der Webseite des DCLs, auf der die Ergebnisse eines Jobs stehen. An diese Url wird der String, der das Ticket repräsentiert, angehängt.
  - pollingRate: Aktualisierungsrate des Job Explorers und des Output Fensters in Millisekunden. Standardwert ist 3000, für langsamere Modemverbindungen empfiehlt es sich den Wert zu erhöhen. Der Minimalwert ist 1000, um zu verhindern, dass ununterbrochen Anfragen an das DCL geschickt werden.
  - cameras: Für jeden Eintrag „camera“ innerhalb dieses Datenelements wird im Menü DCL ein Eintrag erzeugt, der die entsprechende Url in einem externen Fenster anzeigt.

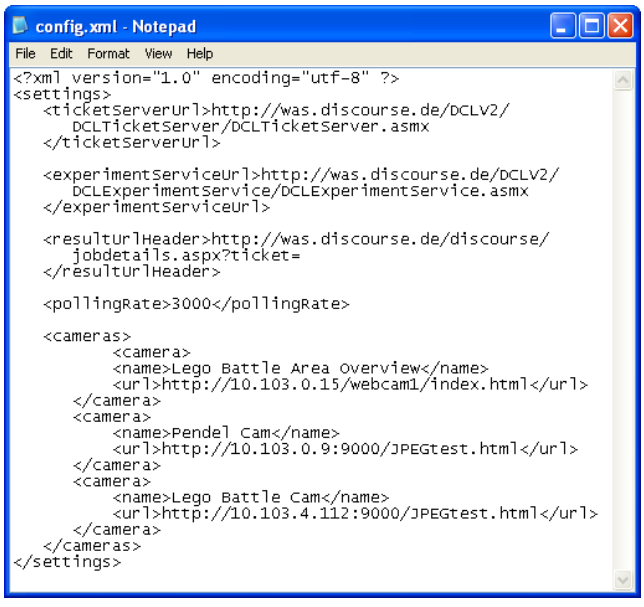


**Abbildung 5: Editieren der config-Datei**

- Nach dem Speichern der Konfigurationsdatei kann Visual Studio gestartet werden. Nun sollte der Menüeintrag DCL vorhanden sein. Um den Benutzernamen und das Passwort des DCLs einzugeben, wählen sie im Menü DCL den Eintrag SETTINGS... (siehe Abbildung 6).

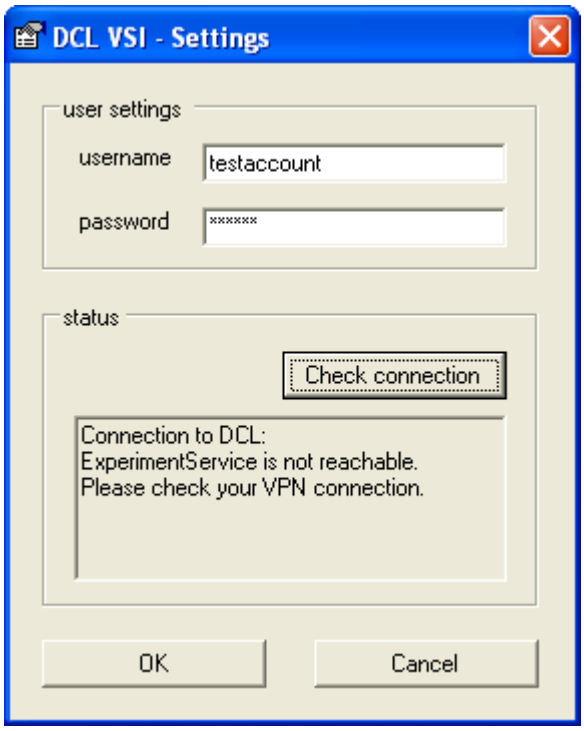


**Abbildung 6: Einstellen der Benutzerdaten des DCLs**

- Mit dem Button CHECK CONNECTION kann die Verbindung zum DCL mit den eingegebenen Daten getestet werden. Hierfür muss die VPN-Verbindung aufgebaut sein. Nachdem mit OK bestätigt wurde, werden die Benutzerdaten in der Registrierungsdatenbank gespeichert und, wenn eine Verbindung besteht, die vorhandenen Jobs im Job Explorer angezeigt.

## 4.2. Senden eines Jobs

Zum Senden einer Codedatei muss diese im Visual Studio geöffnet und das aktuell markierte Dokument sein. Im Menü DCL kann nun z. B. der Eintrag SEND TO LEGO_ROBOTS gewählt werden oder der entsprechende Eintrag in der Auswahlbox in der Symbolleiste. Nach erfolgreichem Senden wird der aktuelle Status im Output Fenster angezeigt und in regelmäßigen Abständen (siehe Kapitel 4.1) aktualisiert.

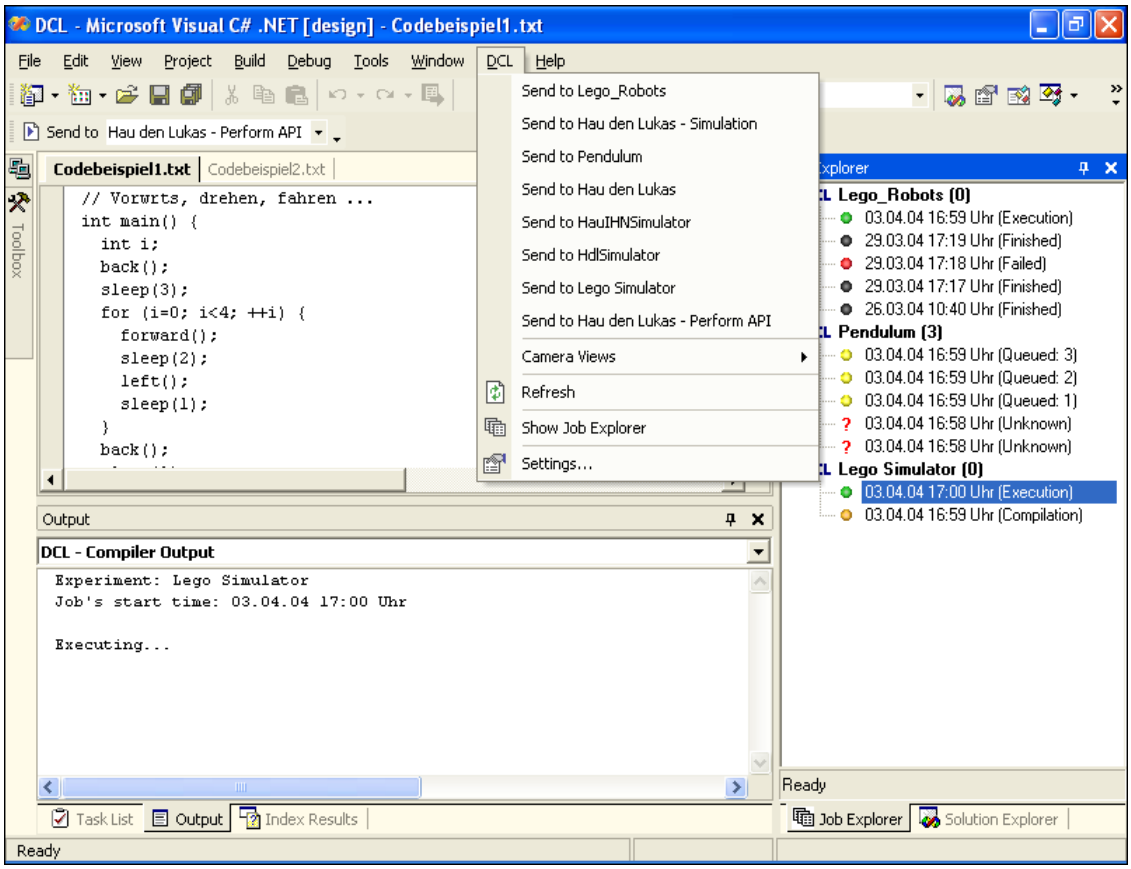


**Abbildung 7: Menü DCL mit aktuellem Status im Output Fenster**

Falls das Codedokument nicht das aktuell markierte ist, so meldet DCL VSI den in Abbildung 8 dargestellten Fehler. Mit einem einfachen Klick ins entsprechende Dokument kann die zu sendende Datei ausgewählt werden.


**Abbildung 8: Fehlermeldung bei falsch markiertem Dokument**

## 4.3. Verwalten der Jobs

Das Kernstück des Add-ins ist der Job Explorer. Hier werden alle Jobs eines Benutzers angezeigt und können verwaltet werden. Sollte der Job Explorer nicht angezeigt werden, so kann er mit dem Eintrag SHOW JOB EXPLORER im Menü DCL in den Vordergrund geholt und beliebig in Visual Studio platziert werden. Mit dem Eintrag REFRESH werden bei bestehender VPN-Verbindung dem Menü DCL Einträge für das Senden von Jobs und das Anzeigen von Kameras hinzugefügt. Zusätzlich wird mit REFRESH die Baumstruktur der Experimente im Job Explorer aufgebaut oder aktualisiert, falls diese schon angezeigt werden.

Im Job Explorer repräsentiert jeder Oberknoten ein Experiment, dessen Anzahl an Jobs in der Warteschlange in Klammern angezeigt wird. Jeder Job wird durch einen Unterknoten zum jeweiligen Experiment dargestellt. Datum und Uhrzeit identifizieren den Job, dessen Status in Klammern oder am entsprechenden Icon zu erkennen ist (siehe Abbildung 9).

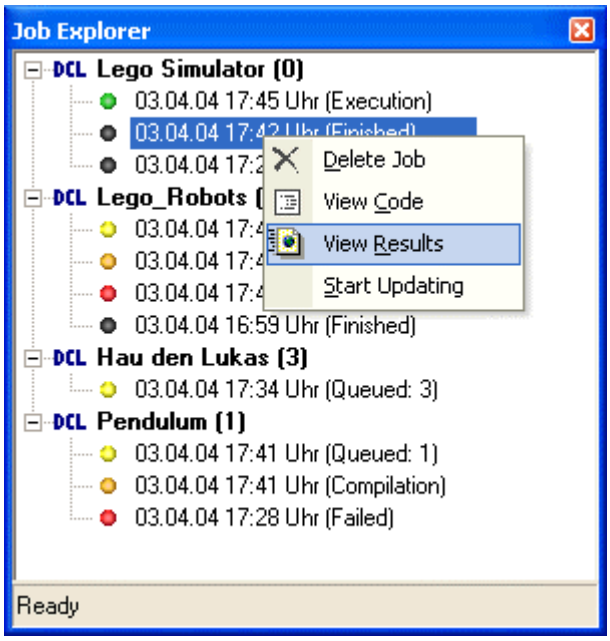


**Abbildung 9: Job Explorer mit Kontextmenü**

Im Kontextmenü des Job Explorer können folgende Aktionen, je nach Zustand des Jobs ausgeführt werden:

- CANCEL JOB: Im Zustand Queued und Compilation kann der Job abgebrochen werden.
- DELETE JOB: Ist der Job in einem Endzustand, so kann er aus dem DCL gelöscht werden, indem das zugehörige Ticket sowohl beim Experimentservice als auch beim Ticketserver gelöscht werden.
- VIEW CODE: Zeigt den Code, der zum DCL geschickt wurde in einem neuen Textfenster an. Diese Datei kann bei Bedarf lokal gespeichert oder aber beim Beenden von Visual Studio wieder verworfen werden.
- VIEW RESULTS: Zeigt die Ergebnisseite, die vom DCL generiert wurde. Zusätzlich wird im Output Fenster eventuell vorhandener Compileroutput und der Result Code angezeigt, wie in Abbildung 7 nach dem Senden eines Jobs.
- STOP UPDATING: Der Status der im Job Explorer angezeigten Jobs, die sich nicht in einem Endzustand befinden, wird in regelmäßigen Abständen aktualisiert. Dies kann mit dem Eintrag STOP UPDATING ausgeschaltet werden.
- START UPDATING: Startet das Aktualisieren des Job Explorers wieder.

Bei einem Doppelklick auf einen Job wird ebenfalls der Code in einem Textfenster angezeigt und zusätzlich der aktuelle Status mit eventuell vorhandenem Compileroutput in das Output Fenster geschrieben.

## 4.4.Anzeigen der Kameras

Für jede in der Datei config.xml angegebene Kamera (siehe Abschnitt 4.1) wird im Untermenü CAMERA VIEWS ein Eintrag erzeugt. Beim Auswählen öffnet sich ein neues Fenster in dem die Url der Kamera angezeigt wird (siehe Abbildung 10).

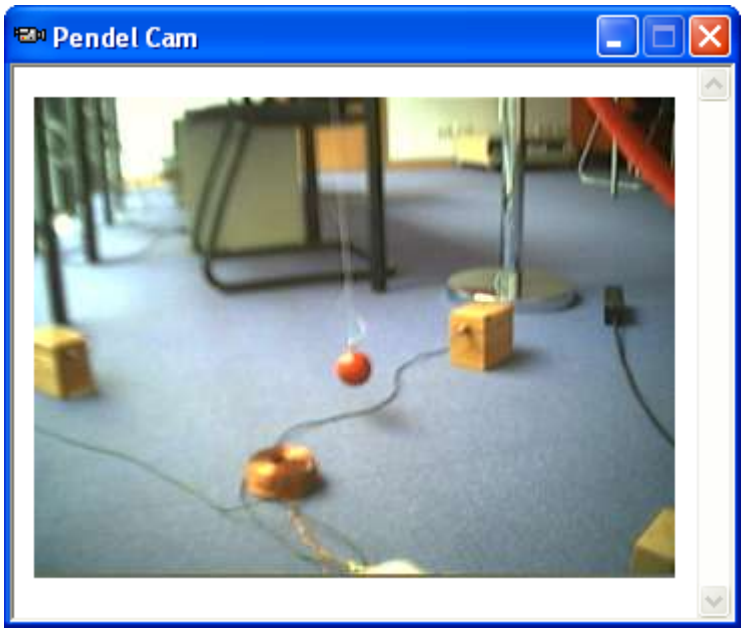


**Abbildung 10: Anzeigen einer Kamera**

# 5. Fazit

Alles in allem war der tatsächliche Aufwand, gegenüber dem zu Beginn geplanten, doch viel höher. Nachdem der Button zum Senden ans DCL und der in der Aufgabenstellung geforderte Konfigurationsdialog noch relativ schnell implementiert waren, erforderte die Anzeige der Ergebnisse eines Jobs mehr Aufwand. Es musste realisiert werden, dass der Benutzer einen Job auswählen kann, um dessen Ergebnisse darzustellen. Die damit verbundene Einführung des Job Explorers, mit all seinen zusätzlichen Features, brachte die meiste Arbeit. Dies war jedoch notwendig, um die Aufgabenstellung vollständig zu erfüllen.

Trotz des unterschätzten Aufwands brachte die Programmierarbeit eine Menge Erfahrung und viele neue Erkenntnisse. Die Einarbeitung in die Schnittstellen von Visual Studio .NET war, wenn auch zum Teil etwas mühsam, sehr interessant. Es waren die vielen kleinen Dinge, die einiges an Zeit kosteten.

# Literaturverzeichnis und Urls


[1] Kemp Brown, “Automation and Extensibility Overview”, Visual Studio .NET Technical Articles, Februar 2002

[2] http://www.vsipdev.com/techinfo/vsipcontents.aspx

[3] Brian A. Randall, “Visual Studio Tools for the Microsoft Office System”, Technical Articles, April 2003

[4] Kemp Brown, “Frequently Asked Questions About Visual Studio .NET Automation”, Technical Articles, Juni 2003

[5] John Robbins, “Google from Visual Studio .NET”, MSDN Magazine, November 2003

[6] Distributed Control Lab V2 Reference