JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Konfigurationswerkzeug „Plugin-Explorer“ für die Plugin-Plattform Plux.NET

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Andreas Gruber, 0655500

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
Mag. Reinhard Wolfinger

Linz, Jänner 2010

Vorwort

Diese Bachelorarbeit beschreibt die Funktionalität und Implementierung des Konfigurationswerkzeugs Plugin-Explorer, kurz Explorer, für die Plugin-Plattform Plux.NET. Mit dem Explorer kann ein Benutzer ein Programm aus Plugins zusammenstecken, ohne dass er dabei programmieren muss. Der Explorer zeigt die Komponenten eines Programms als Graph an. Dynamische Kontextmenüs enthalten die Funktionen zum Konfigurieren des Programms.

Die Kapitel 1 und 2 beschreiben die ursprüngliche Aufgabenstellung und den Funktionsumfang des Explorers. Danach folgt ein Kapitel über die Bedienung des Programms anhand eines konkreten Beispiels. In den Kapiteln 4 und 5 folgt der technische Teil. Dabei wird als erstes der grundsätzliche Aufbau des Explorers beschreiben und danach wird auf wichtige Details der Implementierung einzelner Komponenten eingegangen. Abgeschlossen wird die Arbeit mit einem persönlichem Fazit und einem Ausblick, wie man das Programm weiterentwickeln könnte.

Ich bedanke mich bei Herrn Mag. Reinhard Wolfinger und bei Herrn Dipl.- Ing. Markus Jahn für die Unterstützung und das zu Stande kommen dieser Arbeit.

Inhaltsverzeichnis

Kapitel 1

Aufgabenstellung

Plux.NET ist eine Plugin-Plattform für Microsoft.NET und ermöglicht erweiterbare Programme, bestehend aus einem ultradünnen Kern und einer Sammlung von Erweiterungen. Plugins können zur Laufzeit in Steckplätze des Kerns oder in Steckplätze anderer Erweiterungen eingesteckt werden. Weiterführende Informationen zu Plux.NET finden Sie auf der Projektseite [2].
Für Plux.NET gibt es das Visualierungswerkzeug HotViz, das die Architektur einer Plugin-Anwendung sowie deren Änderungen zur Laufzeit als Graph anzeigt. HotViz zeigt geladene Erweiterungen, offene Steckplätze und in welche Steckplätze Erweiterungen eingesteckt sind.

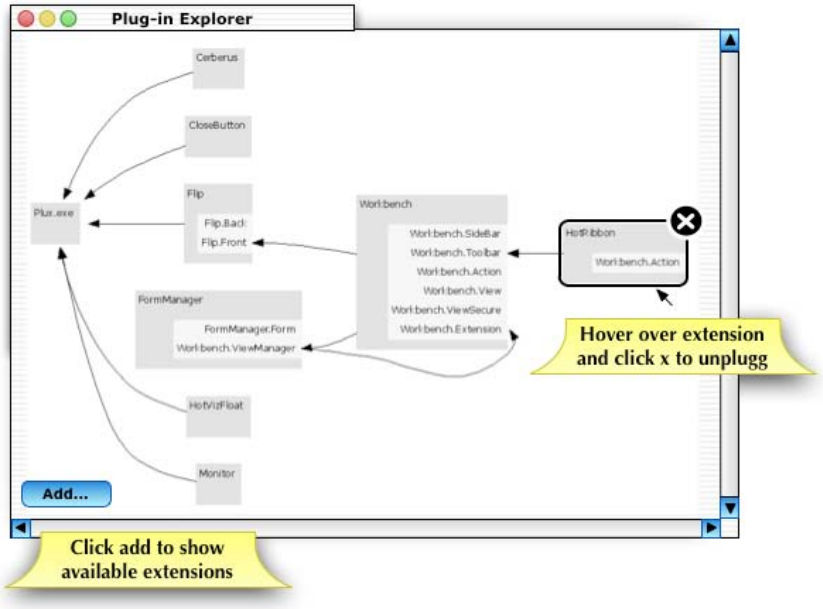


Abbildung 1.1: Entwurf der Benutzerschnittstelle des Plugin-Explorers

Das Visualisierungswerkzeug HotViz soll zu einem Konfigurationswerkzeug umgebaut werden, mit dem die Architektur nicht nur angezeigt, sondern auch konfiguriert werden kann. Ziel dieser Arbeit ist Design und Implementierung des Konfigurationswerkzeugs "Plugin-Explorer" mit folgenden Leistungsmerkmalen:

- Multiple Views: Ansichten für geladene Plugins/Extensions (ein Plugin kann mehrere Extensions enthalten) oder verfügbare (entdeckte) Plugins/Extensions.

- Zoom und Autoscroll: Stufenloses Zoomen der Ansicht. Bei Änderungen automatisches Scrollen zum Punkt der Änderung.
- Drag and Drop Plugging, Guides: Einstecken von neuen Plugins durch Drag and Drop aus einem Komponenten-Repository in die View. Hervorhebung der passenden Steckplätze beim Ziehen (siehe Abbildung 1.2).

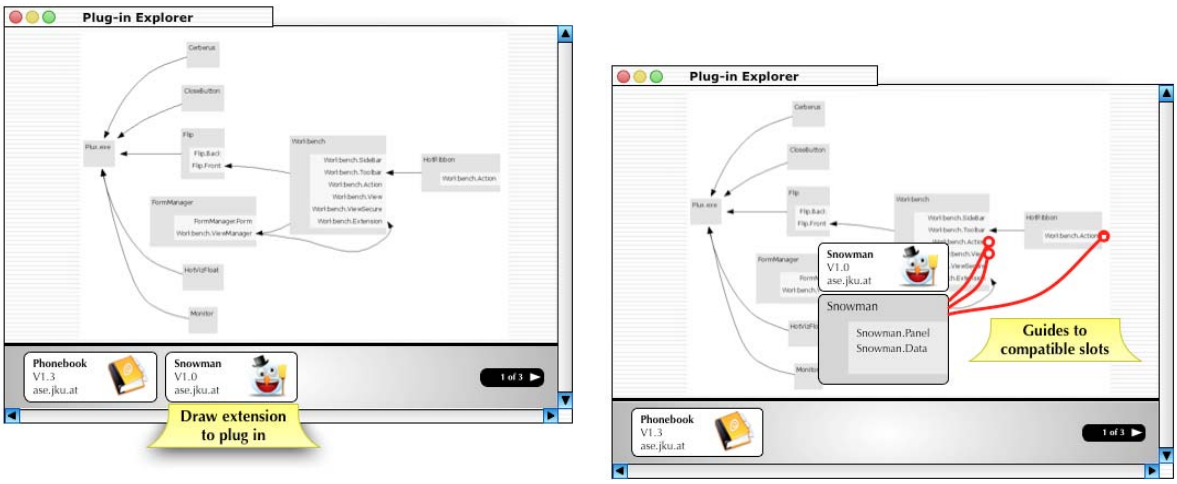


Abbildung 1.2: Entwurf für Drag and Drop Plugging

- Unplugging: Gezieltes Ausstecken aus einzelnen Steckplätzen beziehungsweise vollständiges Entladen des Plugins (siehe Abbildung 1.1).
- Hot Updating: Aktualisieren von Plugins mit neuen Versionen durch Drag and Drop zur Laufzeit.
- Live Statistics: Anzeige von Architekturmetriken, Anzahl der aktuell entdeckten/-geladenen Plugins/Extensions, Anzahl der insgesamt seit Programmstart entdeckten/geladenen/entladenen Plugins/Extensions, Anzahl der Steckplätze, usw.

Der Plugin-Explorer ist mit Visual Studio 2005 in C# mit Windows Forms 2.0 zu implementieren. Informationen zu Windows Forms 2.0 und Bücher, die zur Unterstützung bei der Programmierung verwendet wurden, sind im Literaturverzeichnis aufgelistet.
Diese Aufgabenstellung ist aus [1] übernommen worden. Sie wurde in der Phase des Prototypings adaptiert und auf die Einschränkungen und Eigenheiten von Windows Forms angepasst. Der endgültige Funktionsumfang des Explorers, und somit auch dieser Arbeit, ist Kapitel 2 beschrieben. Hot Updating wurde aus Gründen der Komplexität gestrichen, aus Live Statistics wurde ein eigenständiges Projekt (Plux-Metrix).

Kapitel 2

Funktionsumfang

Der Plugin-Explorer ist ein grafisches Konfigurationsprogramm für Plux.NET. Es ermöglicht eine nahezu vollständige Kontrolle über ein Plux.NET Programm und der Benutzer kann aktiv das Programm manipulieren. Der Explorer kann auch so konfiguriert werden, dass er sich passiv verhält und das System nur visualisiert.

Plux.NET Integration

Der Explorer nutzt nahezu den gesamten Funktionsumfang des Plux.NET Frameworks für das Konfigurieren und Erstellen von plugin-basierten Anwendungen. Es können sämtliche Properties der einzelnen Komponenten verändert werden. Das heißt für jede Komponente, wie zum Beispiel für Slot oder Extension, stehen alle von Plux.NET zu Verfügung gestellten Methoden und Eigenschaften auch im Explorer zu Verfügung.

Dynamische Menüs

Die gesamte Funktionalität des Explorers befindet sich in Kontextmenüs. Jedes einzelne Kontextmenü wird dynamisch aufgebaut und bietet somit immer die aktuellsten Auswahlmöglichkeiten. Enthalten Menüeinträge keine Unterpunkte oder sie sind zum Zeitpunkt des Aufrufs nicht möglich, so wird der entsprechende Eintrag deaktiviert. Sind bei einem Unterpunkt mehrere Aktionen gleichzeitig möglich, gibt es auch immer eine Möglichkeit alle diese Aktionen gemeinsam auszuführen.

Anpassbare Ansichten

Grundsätzlich wird bei Plux.NET mit Typen (ExtensionTypeInfo), Instanzen (ExtensionInfo), Plugs (PlugTypeInfo und PlugInfo) und Slots (SlotTypeInfo und SlotInfo) gearbeitet. Diese Komponenten werden im Explorer in Form eines Extension-Graphs dargestellt. Da auch noch zusätzlich Parameter und Properties der einzelnen Komponenten existieren, kann der Graph sehr komplex und unübersichtlich werden. Aus diesen Gründen ist die Ansicht des Graphs, im speziellen die der Knoten, anpassbar. Jede dieser Einzelkomponenten kann angezeigt oder ausgeblendet werden, somit kann immer genau die Ansicht erzeugt werden, die für die benötigte Aufgabe die Richtige ist. Zudem werden Typen und Instanzen unterschiedlich dargestellt, sodass eine Unterscheidung am ersten Blick möglich ist.

Typenmanagement

Extension Types kommen im Explorer in zwei verschiedenen Ausprägungen vor. Entweder sind sie mit einem Slot verbunden oder sie sind frei und ungebunden im Graph. Es gibt verschiedene Möglichkeiten diese Typen zu suchen und finden. Typen können gruppiert angezeigt, oder auch versteckt werden. Zusätzlich gibt es auch eine Möglichkeit ungebundene Typen vollständig zu entfernen. Wenn von einem Typ eine Instanz erstellt wird, so wird diese direkt beim Typ angezeigt und es kann damit unmittelbar weitergearbeitet werden.

Dynamischer Graph

Der Graph ist durch den Benutzer vielfältig manipulierbar, das heißt es können einzelne Knoten und Subgraphen verschoben werden. Markierte Knoten können mit der Tastatur in kleinen Schritten verschoben werden und es kann in oder aus dem Graphen gezoomt werden. Einzelne Subgraphen können unabhängig vom Gesamtgraph neu angeordnet werden. Weiters werden geschlossene Slots und selektierte Plugs graphisch dargestellt.

Unterschiedliche Bearbeitungsmodi

Der Explorer kann in einem automatischen oder manuellen Modus verwendet werden. Im automatischen Modus können einfache Aufgaben ausgeführt werden und die Komplexität des Programms bleibt verborgen. Im manuellen Modus können alle Aktionen vom Benutzer selbst ausgeführt werden, das heißt der Benutzer hat volle Kontrolle über Plux.NET und sieht auch gesamten Funktionsumfang des Explorers.

Plux.NET Framework

Damit sich der Explorer nahtlos in das Plux.NET Framework einfügen kann, ist der Explorer selbst auch mit Plux.NET entwickelt. So wird das dauerhafte Speichern der gesamten Einstellungen von Plux.NET übernommen. Über einen von Plux.NET zu Verfügung gestellten Eigenschaften-Dialog können viele Einstellungen geändert werden. Der Explorer kann mittels spezieller Plugs sowohl als eigenständiges Fenster, als auch als Subfenster in der Workbench dargestellt werden.

Erweiterbarkeit

Der Explorer nutzt Plux.NET, um selbst erweiterbar zu sein. Es wird ein Slot angeboten, in den neue Algorithmen zum Anordnen der Knoten des Graphs angesteckt werden können. Zusätzlich ist auch das Benachrichtigungssystem austauschbar, das heißt Benachrichtigungen können nicht nur am Bildschirm ausgegeben werden, sondern zum Beispiel auch in eine Datei geschrieben werden.

Toolbar

Alle verfügbaren Plugins werden in einer optionalen Toolbar gruppiert. Innerhalb eines Plugins werden wiederum alle Typen des Plugins angezeigt. Mittels der Toolbar können gesamte Plugins oder einzelne Typen dem Graphen hinzugefügt werden. Die Toolbar verfügt über die selben Möglichkeiten zum Scrollen wie der Graph und passt ihre Darstellung an den verfügbaren Platz an.

Kapitel 3

Bedienung

3.1 Benutzerschnittstelle

Um die Grundzüge des Explorers zu verstehen wird im ersten Teil dieses Kapitels auf die allgemeinen Teile des Programms eingegangen, danach werden einzelne Funktionen anhand eines Beispiels näher beschrieben. Zu Beginn wird Plux.NET zusammen mit der Extension Cerberus, einigen wichtigen Contracts und dem Explorer gestartet. Der Explorer verwendet standardmäßig eine Extension für das Layout und für die Benachrichtigungen, damit er von Beginn an voll einsatzfähig ist. Falls auch Einstellungen geändert werden sollen, wird zusätzlich die PropertyDialog Extension benötigt. Beim Start werden keine Einstellungen von Plux.NET geändert, sodass der Explorer anfänglich transparent agiert. Der erste Blick auf das Programm zeigt, dass es keine Menüleiste oder Buttons gibt, alle Funktionen sind in den Kontextmenüs der jeweiligen Komponenten zu finden, zusätzlich stehen auch programmweite Tastaturbefehle zu Verfügung.

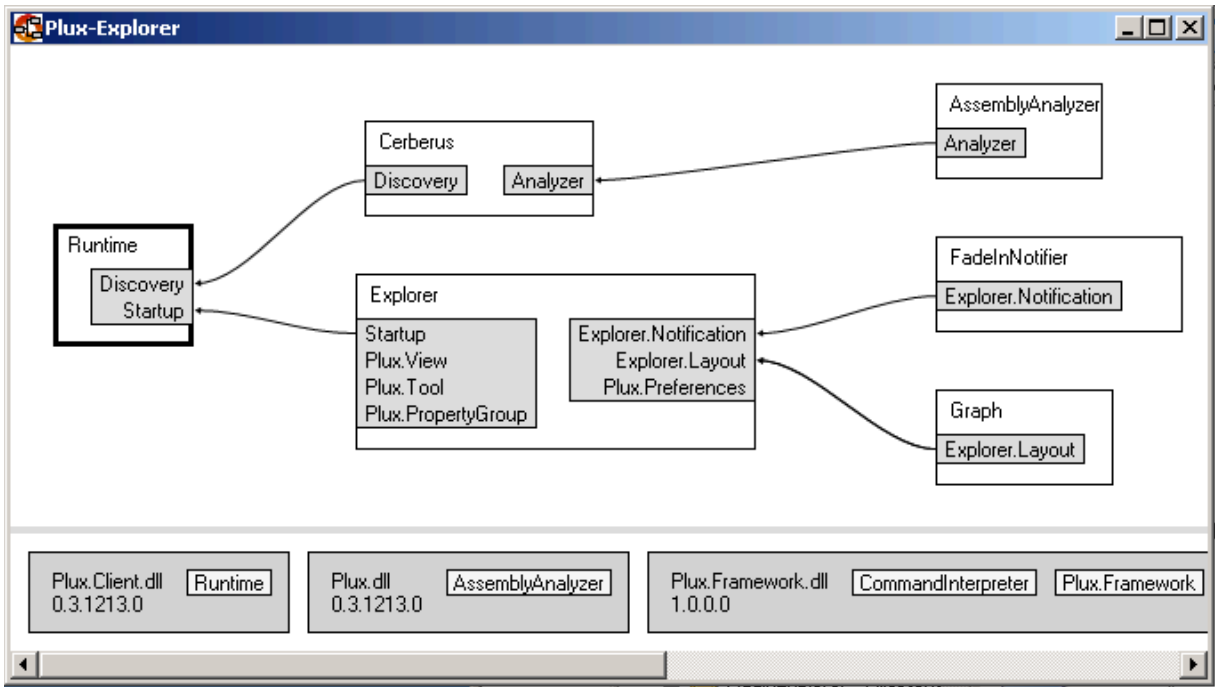


Abbildung 3.1: Startbildschirm des gestarteten Explorers

Abbildung 3.1 zeigt den Explorer nach dem Start. Im oberen Bereich ist ein Graph mit den derzeitigen Extensions, sowie deren Slots und Plugs, zu sehen. Das Aussehen der Knoten ist von den gewählten Einstellungen abhängig. Im unteren Bereich sind alle verfügbaren Plugins eingeblendet. Beide Bereiche können entweder mit dem Mausrad oder den Cursortasten gescrollt werden. Zum Scrollen muss mit der Maus in den jeweiligen Bereich geklickt werden, um ihn zu aktivieren. Wird ein einzelner Knoten ausgewählt, so kann dieser mit den Cursortasten oder mittels der Maus bewegt werden. Wird zusätzlich die Umschalt-Taste gedrückt, so kann mit der Maus ein ganzer Teilgraph verschoben werden. Über das Kontextmenü des Graph-Bereiches kann ein beliebiger Zoom eingestellt werden. Das Kontextmenü des Graphs beinhaltet auch sämtliche Funktionen zur Anpassung der Ansicht. Es können Plugins, Extension Types, Slots, Parameter, Plugs und Properties ein- und ausgeblendet werden. Extensions können als einzige Komponente nicht ausgeblendet werden. Für die Pfeile zwischen den Knoten ist die Layout-Extension zuständig. Abbildung 3.2 zeigt den Explorer in einer angepassten Ansicht.

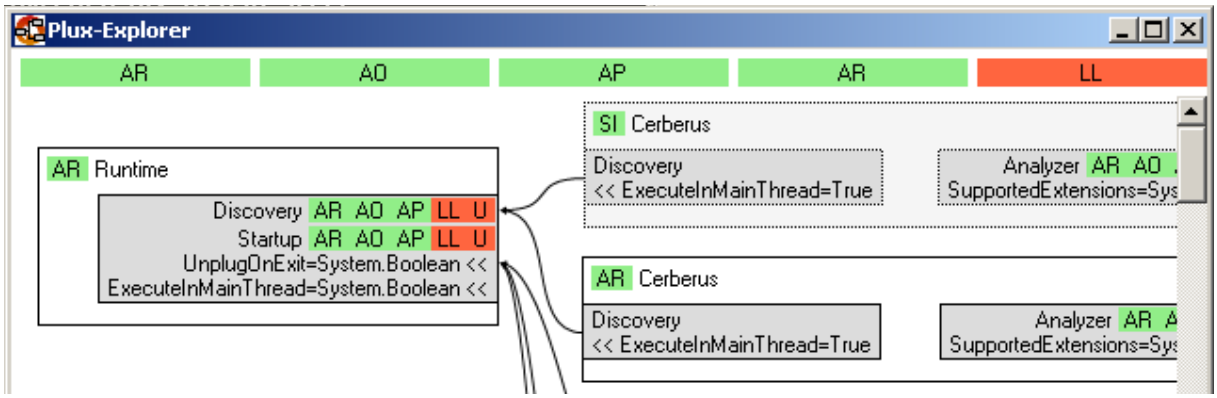


Abbildung 3.2: Ansicht mit aktivierten Extension Types, Properties und Parameter

Die Extension Types sind im Graph immer gepunktet umrandet und zusätzlich in einem anderen Farbton gehalten, damit sie auf den ersten Blick zu normalen Extensions unterschieden werden können. Die Properties der einzelnen Komponenten sind immer direkt bei der Komponente über Buttons zu ändern. Grün symbolisiert ein aktiviertes Property, rot eine deaktiviertes Property. Die Runtime, Slots und Extensions besitzen solche umschaltbaren Properties. In der Abbildung 3.2 sind auch die Parameter eingeblendet. Bei den Slots werden die Parameterdefinitionen und deren Typ angezeigt, graphisch werden diese Parameter mit << dargestellt. Bei den Plugs wird der Parametername und der Parameterwert angezeigt, im Graph wird >> zur Visualisierung verwendet.
Das Kontextmenü des Graphs beinhaltet noch viele weitere Funktionen. Es können damit die Extension Types aufgelistet und gesucht werden, der Graph kann noch weiter angepasst werden und die Farboptionen können aufgerufen werden. Zu jedem dieser Befehle gibt es einen Tastaturbefehl. Eine wichtige Funktionen dieses Menüs ist das Umschalten des Modus. Die beiden Modi unterscheiden sich dadurch, dass im manuellen Modus in den Kontextmenüs der einzelnen Komponenten die volle Funktionalität zu Verfügung steht. Ursprünglich wurden auch die Properties der Runtime je nach Modus geändert, doch diese Funktionalität wurde entfernt, da sie zu Problemen führte wenn der Explorer mit anderen Plugins betrieben wurde.

3.2 Beispiel

Zu Beginn werden die Properties im Explorer aktiviert und in der Runtime das Property AutoRegister deaktiviert, damit der Explorer die Kontrolle über Plux.NET übernehmen kann. Das Beispiel zeigt nun das Zusammenspiel zwischen dem Explorer, der Workbench und der HelloWorld-Extension. Dazu müssen die beiden Extensions in das Arbeitsverzeichnis des Explorers kopiert werden, diese erscheinen dann sofort in der Toolbar. Das Ergebnis dieser Schritte in Abbildung 3.3 zu sehen.

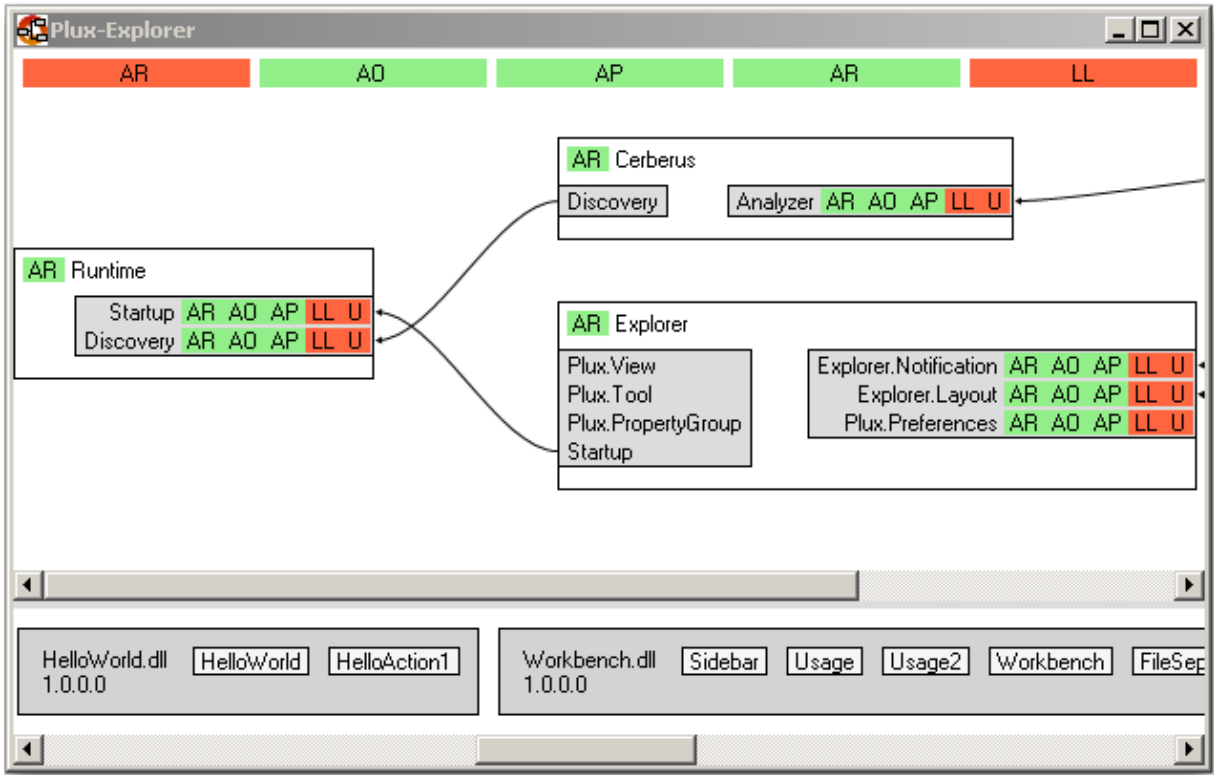
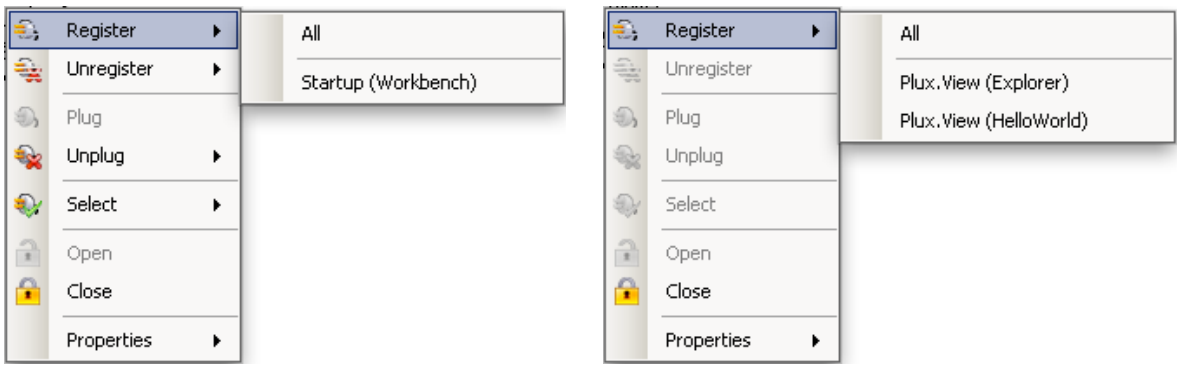


Abbildung 3.3: Explorer mit eingeblendeter Toolbar

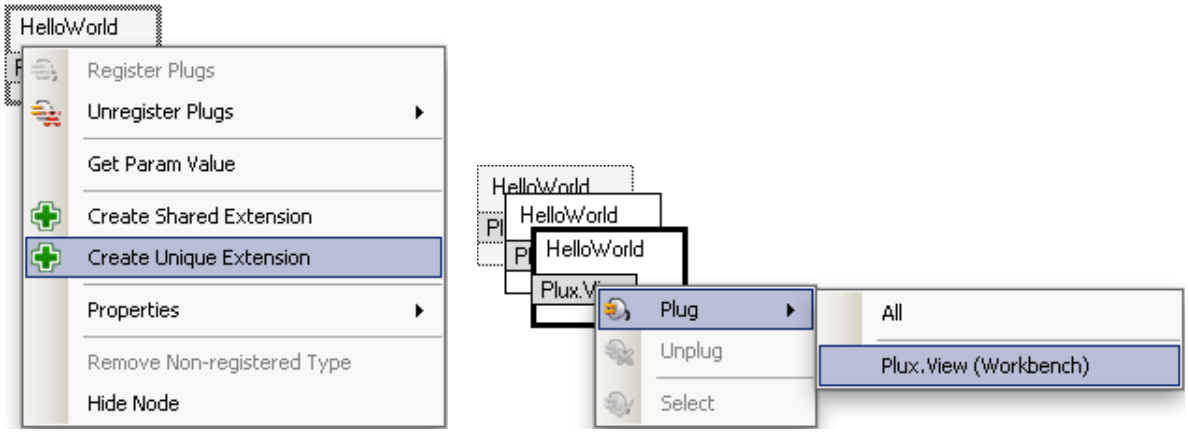
Für die nächsten Schritte muss der Explorer im manuellen Modus sein, da alle Funktionen der Kontextmenüs benötigt werden. Diese Einstellung ist im Kontextmenü des Graphs zu ändern.



(a) Workbench (b) HelloWorld

Abbildung 3.4: Aufrufe zum Registrieren der Extensions Types

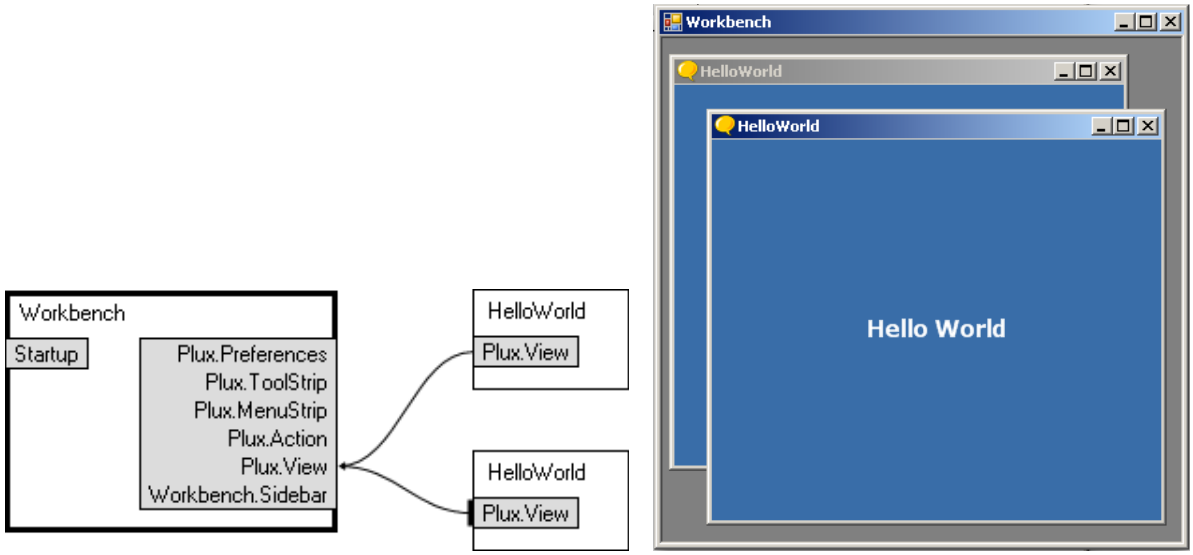
Nun wird die Workbench im Slot Startup der Runtime registriert. Dazu muss der Befehl Register im Kontextmenü (3.4a) des Startup-Slots aufgerufen werden. Dieses Kontextmenü bietet zusätzlich eine Plug- und OpenSlot-Funktion, sowie deren gegenteilige Funktionen. Es kann ein Plug selektiert werden und für den Slot können auch einzelne Properties geändert werden. Die Workbench wird automatisch gepluggt, da das Property AutoPlug aktiviert ist. Im nächsten Schritt (3.4b) wird der HelloWorld Extension Type im Slot Plux.View der Workbench registriert. Da der Explorer die Typen nicht standardmäßig anzeigt, wird der Typ über das Kontextmenü des Graphs gesucht und angezeigt. Von diesem Extension Type müssen nun konkrete Extensions erzeugt werden, diese Funktion ist im Kontextmenü des HelloWorld Extension Types zu finden (3.5a). Das Kontextmenü der Extension Types bietet darüber hinaus Funktionen zum Registrieren von Plugs, es können die Werte der Parameter ausgegeben werden und grafische Optionen sowie Properties geändert werden.



(a) Erstellen (b) Pluggen

Abbildung 3.5: Pluggen von HelloWorld an die Workbench

Für das Beispiel werden mittels CreateUnique-Extension Befehl zwei neue HelloWorld-Extensions erzeugt. Diese werden dann an den Plux.View-Slot der Workbench gepluggt. Diese Funktion lässt sich am Plux.View-Slot der Workbench, dem Plux.View-Plug von HelloWorld sowie bei den jeweiligen Extensions aufrufen. In diesem Fall wird das Plug-Kontextmenü (3.5b) der einzelnen HelloWorld-Extensions dazu verwendet.
Durch das Pluggen der Extensions wird eine Struktur wie in Abbildung 3.6a erzeugt. Die breite Markierung beim Plug der unteren HelloWorld-Extension ist eine Selektion. Über das Kontextmenü des Plux.View-Slots könnte die Selektion auch auf die obere Extension gelegt werden. Eine Selektion ist auch über den entsprechenden Plug möglich. In Abbildung 3.6b würde eine Änderung der Selektion bewirken, dass das zu HelloWorld gehörende Fenster im Vordergrund angezeigt wird. Wenn in der Benutzerschnittstelle der Workbench das zu HelloWorld gehörende Fenster geändert wird, dann wird auch im Explorer die Selektion geändert.



(a) Struktur (b) Benutzerschnittstelle

Abbildung 3.6: Struktur und Benutzerschnittstelle der Workbench

Plux.NET ist so aufgebaut, dass es in unterschiedlichen Komponenten Funktionen gibt, die zum selben Ergebnis führen. So kann zum Beispiel das Pluggen einer Extension sowohl vom Plug, als auch vom Slot aus aufgerufen werden. Aus diesem Grund gäbe es noch weitere Wege, dieses Beispiel zu beschreiben. Zusätzlich könnten noch mehr Properties deaktiviert werden, dies hätte zur Folge dass die Runtime weniger Aufgaben automatisch erledigt und der Bediener des Explorers mehr Schritte erledigen muss, um zum gewünschten Endergebnis zu kommen.

Kapitel 4

Aufbau

In diesem Kapitel werden die Klassenstruktur und die Schnittstellen des Explorers beschrieben. Der Explorer besteht aus 37 Klassen, deshalb ist das Klassendiagramm stufenweise aufgebaut und wird Schritt für Schritt verfeinert. Primär wird das Zusammenspiel und die wichtigsten Methoden der Klassen beschrieben.

4.1 Klassen

4.1.1 Explorer

Der Einstiegspunkt in das Programm ist die Klasse Explorer (Klassendiagramm 4.1).

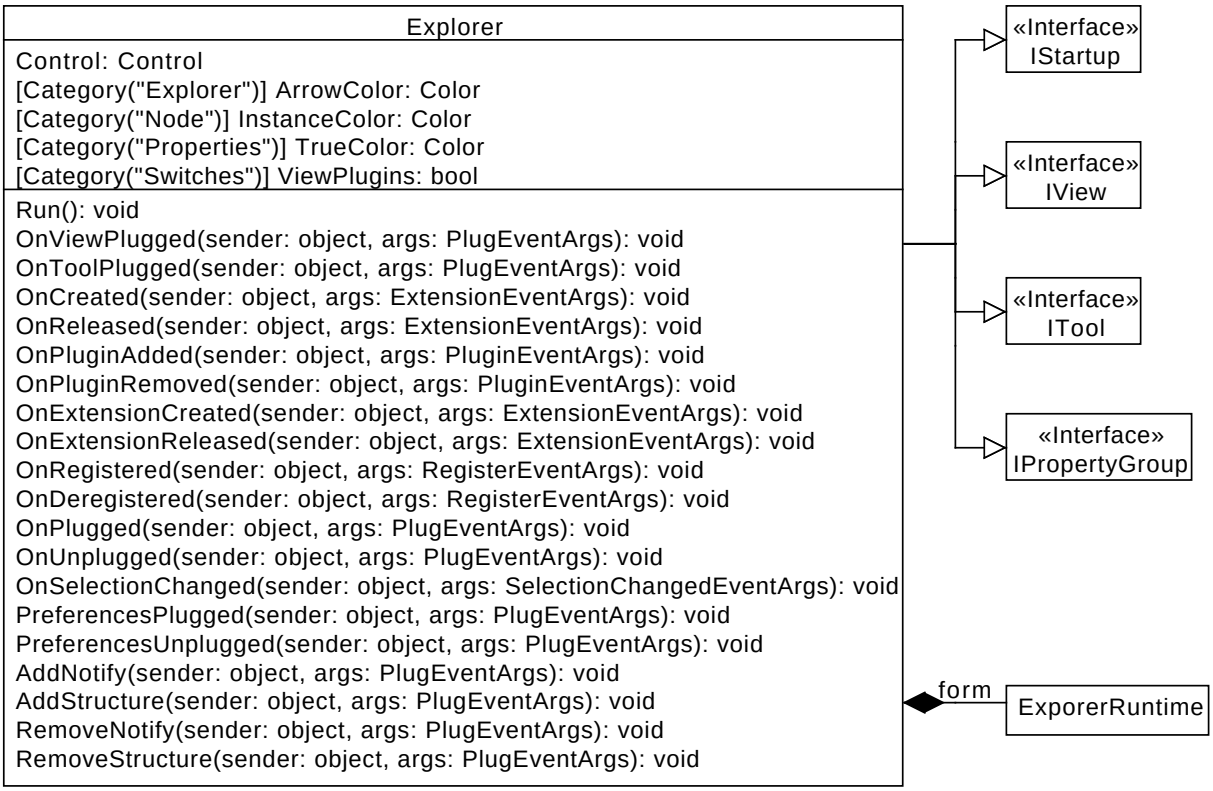


Abbildung 4.1: Klassendiagramm Explorer

Attribut Extension

Zu den Aufgaben der Klasse *Explorer* gehört das Verwalten von Plugs und Slots, sowie das Monitoring von Ereignissen der Runtime. Da der Explorer selbst ein Plux.NET Plugin ist, ist das Attribut *Extension* (Quellcode 4.1) notwendig. Dieses Attribut deklariert die Klasse *Explorer* als Extension. Wenn die Extension erstellt wird, wird die Methode *OnCreated* aufgerufen. Beim Freigeben der Extension kommt die Methode *OnReleased* zum Einsatz. Dabei wird das Fenster versteckt, erst bei einem neuerlichen Erstellen der Extension wird das Fenster wieder angezeigt.

```
[Extension("Explorer",
  OnCreated = "OnCreated", OnReleased = "OnReleased")]
```

Quellcode 4.1: Attribut Extension

Monitor Attribute

Um auf Änderungen im Plux.NET System reagieren zu können, wird eine Vielzahl von Runtime-Ereignissen beobachtet. Damit ein Plux.NET-Ereignis beobachtet werden kann, muss das Monitor-Attribut verwendet werden. In Quellcode 4.2 sind alle verwendeten Monitoring-Funktionen aufgelistet. Es wird auf das Hinzufügen und Entfernen von Komponenten, sowie auf das Registrieren und Pluggen reagiert. Jede dieser Funktionen findet sich auch im Klassendiagramm wieder und diese Funktionen rufen wiederum Funktionen von den Klassen *Graph* oder *Toolbar* auf. Die Toolbar wird von den beiden *OnPlugin*-Methoden verwendet, alle anderen Monitoring-Funktionen hängen mit der Klasse *Graph* zusammen.

```
[Monitor(
  OnRegistered = "OnRegistered", OnPluginAdded = "OnPluginAdded",
  OnUnplugged = "OnUnplugged", OnDeregistered = "OnDeregistered",
  OnPlugged = "OnPlugged", OnPluginAdded = "OnPluginAdded",
  OnPluginRemoved = "OnPluginRemoved",
  OnExtensionCreated = "OnExtensionCreated",
  OnSelectionChanged = "OnSelectionChanged",
  OnExtensionReleased = "OnExtensionReleased")]
```

Quellcode 4.2: Monitor Attribute

Parameter

Plugs können parametrisiert werden, die verwendeten Parameter sowie deren Paramterwerte sind in Quellcode 4.3 angegeben. Die Parameter *ExecuteInMainThread* und *UnplugOnExit* werden für den Plug *Startup* verwendet, die Parameter *OrderIndex*, *Name* und *Kind* werden von den restlichen Plugs verwendet. Je größer der Parameter *OrderIndex* ist, desto weiter unten wird die Extension in einer Liste angeordnet.

```
[ParamValue("ExecuteInMainThread", true)]
[ParamValue("UnplugOnExit", false)]
[ParamValue("OrderIndex", 0.9f)]
[ParamValue("Name", "Explorer")]
[ParamValue("Kind", ViewKind.Tool)]
```

Quellcode 4.3: Plug Parameterwerte

Plug Startup

Der Plug *Startup* ermöglicht es dem Explorer, an den Slot *Startup* der Runtime gepluggt zu werden. Dazu ist ein Attribut (Quellcode 4.4) notwendig und das Interface *IStartup* muss implementiert werden. Das Interface verlangt die Implementierung der Methode *Run*. Diese Methode wird aufgerufen, wenn die Runtime startet.

```
[Plug("Startup")]
```

Quellcode 4.4: Startup Plug

Plug Plux.View

Der Plug *Plux.View* ermöglicht es der Extension seine View, also den Anzeigebereich, in einer anderen Extension anzeigen zu lassen. Dieser Plug benötigt eine Implementierung des Interfaces *IView*. Konkret wird ein *String* mit dem Namen und ein Feld mit einer *Control* implementiert. Bei diesem Steuerelement handelt es sich um den Inhalt des Explorer-Fensters, welcher in einem anderen Fenster angezeigt werden soll. Dieser Plug wird verwendet, damit der Explorer innerhalb der Workbench oder eines anderen Containers angezeigt werden kann. Die Methode *OnViewPlugged* sorgt dafür, dass der Explorer nur in den rufenden Slot gepluggt wird.

```
[Plug("Plux.View", OnPlugged = "OnViewPlugged")]
```

Quellcode 4.5: Plux.View Plug

Plug Plux.Tool

Beim Plug *Plux.Tool* wird, im Gegensatz zu *Plux.View*, der Anzeigebereich innerhalb eines eigenen Fensters angezeigt und nicht innerhalb eines anderen Fensters. Für diesen Plug muss das Interface *ITool* implementiert werden. Wenn der Explorer in einen *Plux.Tool* Slot gepluggt wird, wird er in der Methode *OnToolPlugged* aus allen anderen *Plux.Tool* und *Plux.View* Slots entfernt und nur in den rufenden Slot gepluggt.

```
[Plug("Plux.Tool", OnPlugged = "OnToolPlugged")]
```

Quellcode 4.6: Plux.Tool Plug

Plug Plux.PropertyGroup

Mit dem Plug *Plux.PropertyGroup* können zur Laufzeit Eigenschaften geändert werden. Dazu muss jedes änderbare Feld mit einem Attribut für die Kategorie gekennzeichnet werden. Im Explorer können so Farbeigenschaften und bool'sche Werte geändert werden. Im Klassendiagramm sind die vier verwendeten Kategorien, sowie je ein Beispiel angeführt. In Plux.NET gibt es die Extension PropertyDialog, über welche diese Eigenschaften geändert werden können. Aufgerufen wird diese Extension über das Kontextmenü des Graphs. Es gibt für diesen Plug auch ein Interface, dieses ist jedoch leer.

```
[Plug("Plux.PropertyGroup", AutoPlug = true)]
```

Quellcode 4.7: Plux.PropertyGroup Plug

Slot Explorer.Layout

Dieser Slot erlaubt es, eine Extension für das Anordnen der Knoten des Graphs an den Explorer zu pluggen. Das Interface *ILayout* ist in Punkt 4.2.3 beschrieben. Wird nun eine entsprechende Extension gepluggt, wird die Methode *AddStructure* aufgerufen, in welcher die gepluggte Extension dem Graphen zugewiesen wird. Wird die Extension wieder entfernt, oder wurde nie eine gepluggt, kann der Graph die Knoten nicht anordnen und alle Knoten werden in der linken oberen Ecke positioniert.

```
[Slot("Explorer.Layout",
  OnPlugged = "AddStructure", OnUnplugged = "RemoveStructure")]
```

Quellcode 4.8: Explorer.Layout Slot

Slot Explorer.Notification

Der Slot *Explorer.Notification* wird dazu verwendet, um eine Extension für Benachrichtigungen an den Explorer zu pluggen. Eine entsprechende Extension muss das Interface (Punkt 4.2.2) implementieren. Wenn keine Notification-Extension gepluggt wird, so gibt der Explorer keine Nachrichten aus. Generell gibt der Explorer über diesen Slot Statusmeldungen aus, wenn ein Ereignis auftritt. Fehler werden, wie bei allen anderen Extensions, auf der Kommandozeile ausgegeben.

```
[Slot("Explorer.Notification",
  OnPlugged = "AddNotify", OnUnplugged = "RemoveNotify")]
```

Quellcode 4.9: Explorer.Notification Slot

Slot Plux.Preferences

Der Slot *Plux.Preferences* dient zum Verwalten von Einstellungen. Wird eine Extension gepluggt, welche einen Plug für diesen Slot besitzt, so werden alle Eigenschaften in *PreferencesPlugged* überschrieben und der Explorer verändert sein Aussehen und Verhalten entsprechend. Wird diese Extension wieder entfernt, so werden in der Funktion *PreferencesUnplugged* alle Eigenschaften hinausgeschrieben. Ein typischer Anwendungsfall für diesen Slot ist das Persistieren von Einstellungen, damit beim nächsten Start des Explorers alle Eigenschaften wiederhergestellt werden können.

```
[Slot("Plux.Preferences",
  OnPlugged = "PreferencesPlugged",
  OnUnplugged = "PreferencesUnplugged")]
```

Quellcode 4.10: Plux.Preferences Slot

Das wichtigste Feld der Klasse *Explorer* ist das Feld *form*, darin wird ein Objekt der Klasse *ExplorerRuntime* gespeichert. Diese Klasse ist auch die nächste Stufe im Klassendiagramm und ist im nächsten Punkt beschrieben.

4.1.2 ExplorerRuntime, Graph, Info

Die Klasse *ExplorerRuntime* (Klassendiagramm 4.2) implementiert das Hauptfenster des Explorers und ist von der Klasse *Form* abgeleitet. Dieses Fenster ist in drei Teile aufgeteilt: oben befindet sich der Info-Bereich, in der Mitte der Graph und unten die Toolbar. Der Info-Bereich ist von *Panel* abgeleitet und beinhaltet Buttons, um die Properties der Runtime umschalten zu können. Dieser Bereich ist sichtbar, wenn über das Kontextmenü des Graphs das Anzeigen der Properties aktiviert ist. Über das Kontextmenü ist auch die Toolbar ein- und ausblendbar. Die Toolbar wird gesondert beschrieben, das Klassendiagramm ist in Punkt 4.1.4 zu sehen. Der Graph ist von *ScrollableControl* abgeleitet und kann im Gegensatz zu den anderen Teilen nicht ausgeblendet werden. Zusätzliche besitzen sowohl die Klasse *ExplorerRuntime* als auch die Klasse *Graph* eine Variable *Notify* für eine Extension des Typs *INotification*.

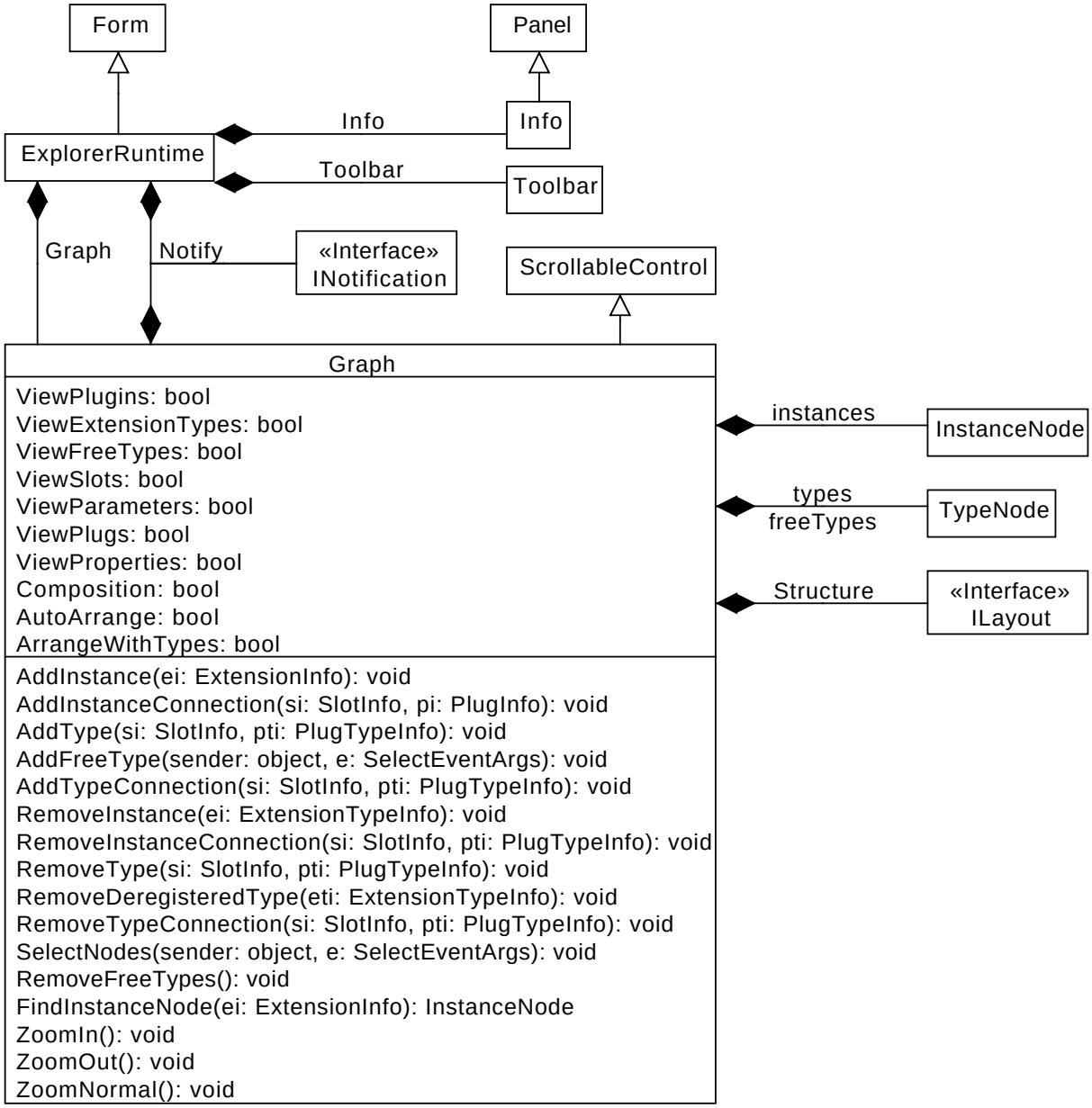


Abbildung 4.2: Klassendiagramm ExplorerRuntime

Die Klasse *Graph* übernimmt die gesamte Verwaltung der Knoten, einzig das Anordnen der Knoten ist ausgelagert. Dazu wird die Variable *Structure* verwendet, in welcher eine Layout-Extension gespeichert wird. Im Zusammenspiel mit dieser Klasse ist der Graph voll funktionsfähig. Ohne diese Extension können die Knoten nicht platziert und angeordnet werden. Auch das Zeichnen der Verbindungen zwischen den Knoten wird durch die Layout-Extension übernommen. Die Klasse *Graph* verwaltet die gesamten Knoten in Listen. Da die Knoten eine komplexe Struktur bilden, sind sie im Punkt 4.1.3 gesondert beschrieben. Für die ExtensionInfos wird eine Liste von *InstanceNodes* geführt. Für die ExtensionTypeInfos sind zwei Listen notwendig, eine für die registrierten Typen und eine für die freien Typen.
Im Klassendiagramm 4.2 sind alle bool'schen Eigenschaften der Klasse *Graph* aufgeführt, über diese Eigenschaften können die verschiedenen Ansichten und das Verhalten des Graphs gesteuert werden. Zusätzlich gibt es auch Farbeigenschaften, doch diese sind nicht explizit aufgeführt, da eigentlich jede sichtbare Farbe geändert werden kann. Die Methoden des Graphs hängen meist mit Monitor-Ereignissen der Plux.NET Runtime zusammen, daraus ergibt sich folgende Zuordnung.

- OnExtensionCreated - AddInstance
- OnExtensionReleased - RemoveInstance
- OnRegistered - AddType und AddTypeConnection
- OnDeregistered - RemoveTypeConnection und RemoveType
- OnPlugged - AddInstanceConnection
- OnUnplugged - RemoveInstanceConnection

Über die Methode *AddFreeType* können über die Toolbar freie *TypeNodes*, also Knoten mit ExtensionTypeInfos ohne einer Zuordnung zu einem Slot, zum Graph hinzugefügt werden. Mit den Funktionen *RemoveDeregisteredType* und *RemoveFreeTypes* können diese *TypNodes* auch wieder entfernt werden. Weiters gibt es noch Funktionen für die Steuerung des Zooms, sowie dem Selektieren und Suchen von Knoten.

4.1.3 BaseNode, BasePlug, BaseSlot, ToggleButton

Die Knoten sind die komplexeste Klassenstruktur im Explorer. Diese Komplexität entsteht dadurch, dass die Knoten die Struktur der Plux.NET Komponenten nachbilden. Ein Knoten ist ein zusammengesetztes Windows-Forms Steuerelement, besteht aus drei Basisteilen und ist in Abbildung 4.3 zu sehen. Mittelpunkt ist die Klasse *BaseNode* mit je einer Liste für *BasePlugs* und *BaseSlots*. Zusätzlich besitzen *BaseNode* und *BaseSlot* eine Liste von *ToggleButtons*. Die Klasse *ToggleButton* dient zur Umschaltung von bool'schen Properties der jeweiligen Plux.NET Komponente. Alle drei Basisklassen, sowie die Klasse *ToggleButton* sind von *Control* abgeleitet und besitzen eine Funktion *CalculateSize*. Diese Funktion berechnet, unter Berücksichtigung der vom Benutzer angepassten Ansicht, die Größe des zu Grunde liegenden Steuerelements. Dabei ist es wichtig, dass jede Basisklasse den Namen der darzustellenden Komponente enthält, da dieser maßgeblich an der Größenberechnung beteiligt ist. Damit die Knoten innerhalb des Graphs richtig angeordnet werden können implementiert jede Basisklasse ein eigenes Interface, der Aufbau

dieser Interfaces ist in Punkt 4.2.1 beschrieben. Die eigentliche Zuordnung von Plux.NET Komponenten zu den Teilen eines Knotens passiert durch Ableiten der Basisklassen. Von *BasePlug* werden die zwei Klassen *TypePlug* und *InstancePlug* abgeleitet. Beide Klassen besitzen ein Feld *Plug*. *TypePlug* speichert eine PlugTypeInfo und *InstancePlug* speichert eine PlugInfo darin. Auch bei *BaseSlot* wird auf die gleiche Art und Weise abgeleitet. Es gibt die Unterklasse *TypeSlot* mit dem Feld *Slot* vom Typ SlotTypeInfo und *InstanceSlot* mit dem Feld *Slot* vom Typ SlotInfo. Auch bei *BaseNode* wird diese Ableitung für ExtensionTypeInfo und ExtensionInfo durchgeführt, es entstehen die Klassen *TypeNode* sowie *InstanceNode*. Wird zum Beispiel ein Knoten für eine ExtensionInfo angelegt, so wird für die Plug- und SlotInfos die jeweilige Liste der *BaseNode* mit den *Instance*-Klassen gefüllt. Zusätzlich werden für Extension- und SlotInfo die benötigten *ToggleButtons* in den Basisklassen abgelegt. Um schnellen Zugriff auf die konkreten Klassen zu erhalten, besitzen sowohl *TypeNode* als auch *InstanceNode* einen Indexer für Plugs und Slots. Die *TypeNode* kann auch eine ExtensionInfo im Feld *CreatedExtension* speichern. Sobald aus einem Typ eine Instanz erzeugt wird, wird dieses Feld belegt und die erzeugte Extension kann beim Typ dargestellt werden. Diese Klasse besitzt auch das bool'sche Feld *Unregistered*, welches wahr ist, wenn der Typ in keinem Slot mehr registriert ist, aber noch im Graph zu sehen ist. Die so genannten freien Typen können über das Kontextmenü des Graphs verwaltet werden.

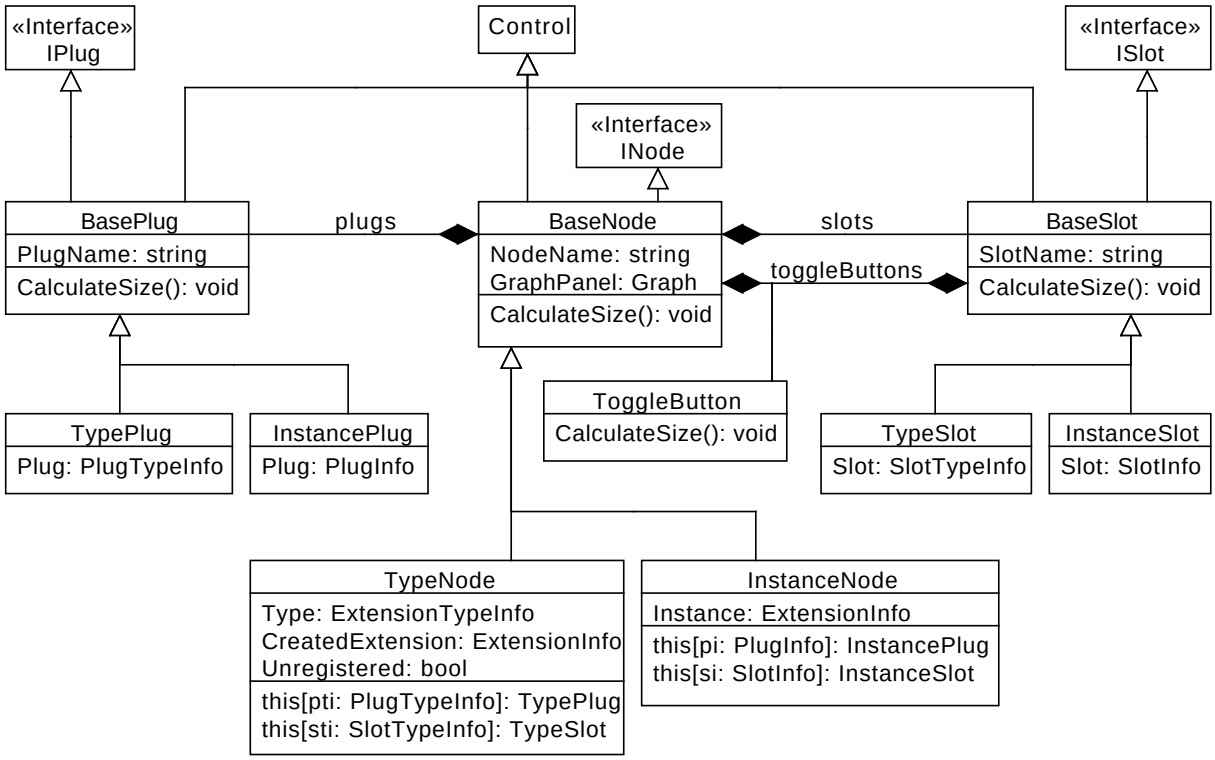


Abbildung 4.3: Klassendiagramm Knoten

4.1.4 Toolbar, ToolbarPlugin, ToolbarType

Die Toolbar verwendet der Explorer zum Auflisten der verfügbaren Plugins und ist, da ein Plugin mehrere ExtensionTypeInfos enthalten kann, stufenweise aufgebaut. Es wird wiederum die Struktur von Plux.NET dargestellt. Der Aufbau der Klassen ist in Abbildung 4.4 dargestellt.

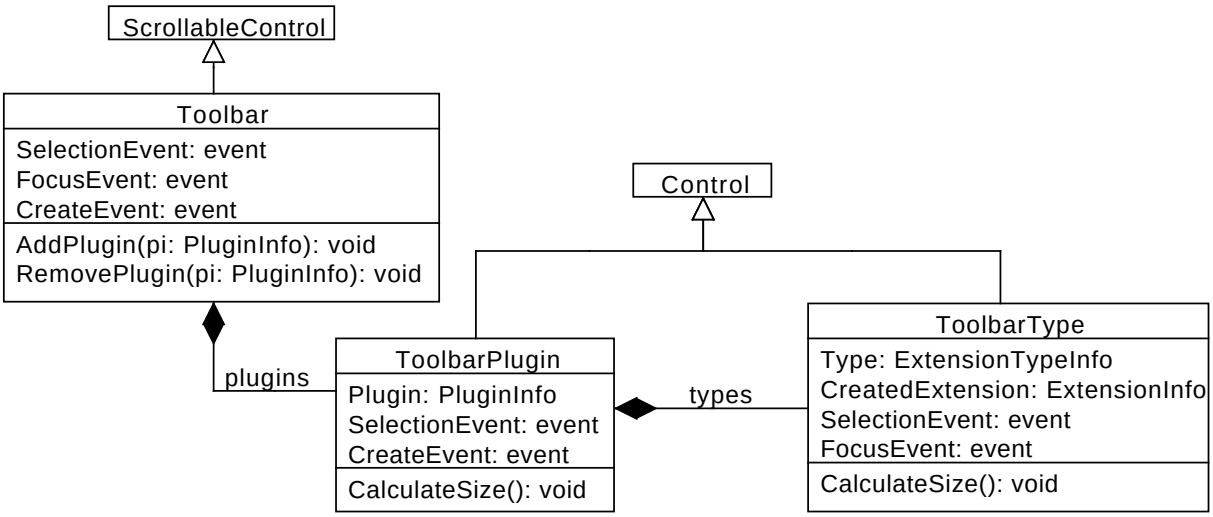


Abbildung 4.4: Klassendiagramm Toolbar

Die Toolbar ist von *ScrollableControl* abgeleitet und verwaltet eine Liste von Elementen der Klasse *ToolbarPlugin*. Die Klasse *ToolbarPlugin* besitzt eine Liste von *ToolbarTypes*. Sowohl *ToolbarPlugin* als auch *ToolbarTyp* sind von der Klasse *Control* abgeleitet und besitzen eine Methode *CalculateSize*, um die Größe der Steuerelemente zu berechnen. Die Methoden *AddPlugin* und *RemovePlugin* erhalten ihre Parameter von der Monitor-Funktionalität und ermöglichen, dass die Toolbar immer den aktuellen Stand der Runtime von Plux.NET entspricht. Zusätzlich werden von der Toolbar Events zu Verfügung gestellt. Das *SelectionEvent* wird ausgelöst wenn sich der Mauszeiger innerhalb eines Plugins oder eines Extension Types befindet. Als Parameter wird eine Liste aller betroffenen ExtensionTypeInfo mitgesendet. Das *FocusEvent* tritt auf, wenn von einem Extension Type eine Instanz erzeugt wird. Es wird die erzeugte Instanz mitgeschickt und auf diese soll der Focus gelegt werden. Das letzte Event ist das *CreateEvent*, es wird gesendet wenn unregistrierte Extension Types erzeugt werden sollen. Alle diese Events werden vom Graph verwendet und entsprechend verarbeitet.

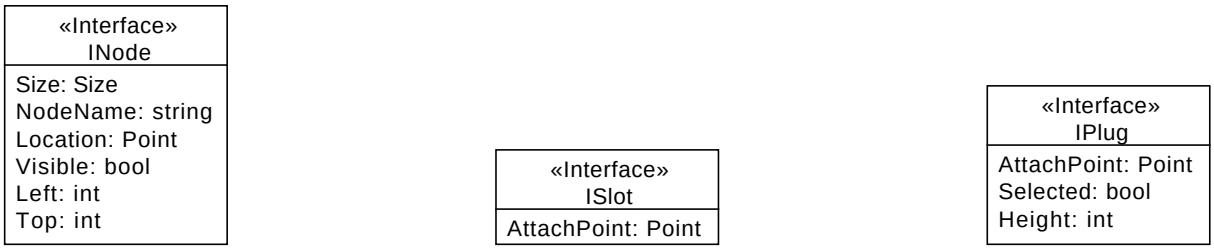
4.2 Schnittstellen

In einer Plux.NET Umgebung zählen nicht nur Interfaces zu den Schnittstellen, viel mehr werden auch Plugs und Slots als Schnittstellen zu anderen Plugins gesehen. Nachdem die Plugs und Slots des Explorers bereits beschrieben wurden, widmet sich dieser Teil nun den Interfaces, welche bereits in den Klassendiagrammen verwendet wurden.

4.2.1 INode, ISlot, IPlug

Die Schnittstellen *INode*, *ISlot* und *IPlug* werden für die Definition eines Knotens im Graph benötigt. Sie legen fest, welche Eigenschaften die Knoten haben müssen, damit das Zeichnen des Graphs möglich ist und die Knoten austauschbar sind. *INode* definiert grafische Attribute eines einzelnen Knotens, dazu zählen zum Beispiel der Name, die Größe und die Position. Eine vollständige Aufstellung der Eigenschaften ist in Abbildung 4.5a zu sehen. Zusätzlich muss eine *INode* das Interface *IComparable<INode>* implementieren, damit die einzelnen Knoten untereinander verglichen werden können. *ISlot* (4.5b) und

IPlug (4.5c) verlangen die Implementierung eines *AttachPoint*. Dieser Punkt wird zum Zeichnen von Verbindungslinien zwischen den einzelnen Knoten verwendet. Zusätzlich definiert *IPlug* noch Felder, damit auch das Selektieren eines Plugs visualisiert werden kann.



(a) INode (b) ISlot (c) IPlug

Abbildung 4.5: Interfaces für Knoten

INode, *ISlot* und *IPlug* werden zusammen in der Klasse *Edge* verwendet. Diese Klasse definiert eine Kante zwischen zwei Knoten. Diese Klasse wurde zum Explorer Contract hinzugefügt, da sie für das Layout benötigt wird. Wie in der Abbildung 4.6 zu sehen, wird der Startknoten mit seinem Startslot und der Endknoten mit seinem Endplug definiert. Für das Anordnen von Knoten ist es wichtig, dass Edges untereinander vergleichbar sind.

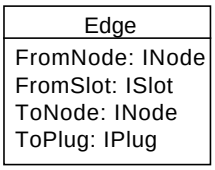


Abbildung 4.6: Klassendiagramm Edge

4.2.2 INotification

INotification definiert eine Schnittstelle, über die der Explorer Nachrichten und Statusmeldungen ausgeben kann. Im Klassendiagramm 4.7 ist das vollständige Interface zu sehen. Über diese Eigenschaften kann der Explorer die Ein-, Anzeige- und Ausblendezeit sowie die Schriftart festlegen. Die Methode *notify* wird zur Ausgabe eines Texts verwendet.

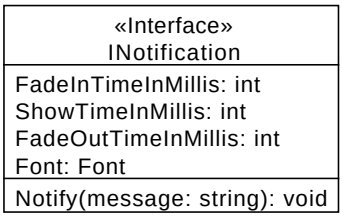


Abbildung 4.7: Interface INotification

Zusätzlich wird auch der Slot (Quellcode 4.11) definiert, über den Notifier an den Explorer gepluggt werden können. Im Zuge dieser Arbeit wurde auch ein Notifier erstellt, welcher beim Start des Explorers immer gepluggt wird.

```
[SlotDefinition("Explorer.Notification")]
```

Quellcode 4.11: Notificatoin Slot Definition

4.2.3 ILayout

Das Interface *ILayout* ermöglicht es, verschiedene Algorithmen an den Explorer zu pluggen welche für das Layout des Graphen zuständig sind. Die Eigenschaften definieren, das Verhalten des Algorithmus. Zum Beispiel kann festgelegt werden ob Knoten automatisch angeordnet werden, oder ob alle Knoten beim Anordnen mit einbezogen werden sollen. Das Interface definiert auch Methoden zum Hinzufügen und Entfernen von Extensions (Instance), Extension Types (Type) und Kanten (Edge). Eine Kante wird erzeugt, wenn Extension gepluggt oder ein Extension Type registriert wird. Im Falle des unpluggen einer Extension oder des unregistrieren eines Extension Types wird die Kante wieder entfernt. Ausgehend von einer Wurzel kann ein Teil des Graphen neu angeordnet oder verschoben werden.

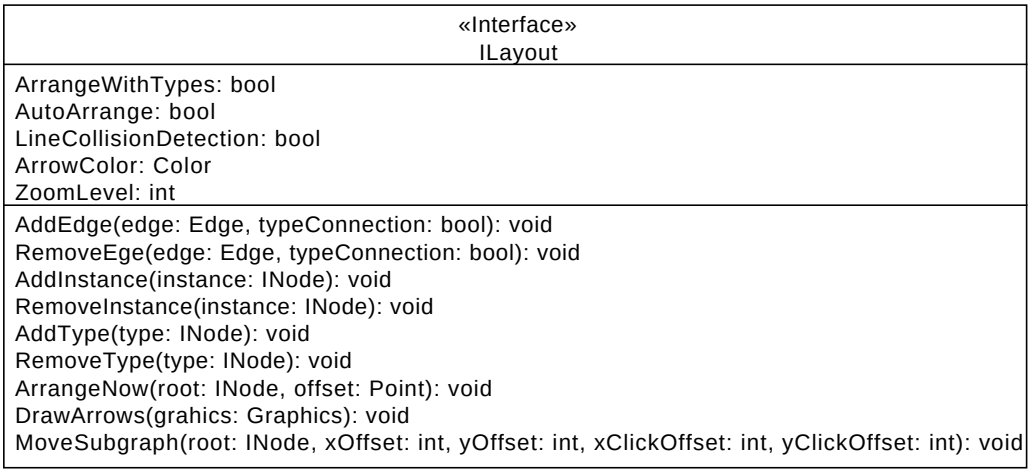


Abbildung 4.8: Interface ILayout

Im Quellcode 4.12 ist die zum Interface zugehörige Slot Definition angegeben. Um also einen eignen Algorithmus zu entwickeln, muss dieses Interface implementiert werden und über diesen Slot kann die Klasse an den Explorer gepluggt werden. Der Explorer kann immer nur einen Algorithmus zum Anordnen der Knoten verwenden, sollte gar keine Layout-Extension an den Explorer gepluggt werden, so bleiben alle Knoten unsortiert in der linken oberen Ecke des Graphs.

```
[SlotDefinition("Explorer.Layout")]
```

Quellcode 4.12: Layout Slot Definition

Der Explorer verfügt über eine Standardimplementierung für diesen Slot, einzelne Teile dieser Implementierung sind im Kapitel 5 genauer beschrieben.

Kapitel 5

Implementierung

Nach dem Aufbau und Zusammenspiel der Klassen wird in diesem Kapitel noch eine Ebene tiefer gegangen. Es werden konkrete Implementierungen und Lösungen verschiedener Problemstellungen behandelt. Zu Beginn werden allgemeine Besonderheiten von Windows Forms beschrieben und danach die Highlights der einzelnen Komponenten des Explorers.

5.1 Windows Forms

Windows Forms ist eine Klassenbibliothek zur Erstellung von Benutzeroberflächen. Diese Klassenbibliothek ist ein Teil des Microsoft .NET Frameworks und bietet eine Vielzahl von vorgefertigten Steuerelementen an. Weiterführende Informationen sind unter [3], [5] und [7] zu finden

Darstellung von benutzerdefinierten Steuerelementen

Anwendungen mit Windows Forms werden üblicherweise aus vorgefertigten Steuerelementen wie Buttons und Textfelder zusammengesetzt. Natürlich gibt es die Möglichkeit eigene Elemente zu entwickeln und diese selbst zu zeichnen [4], dies kann jedoch zu Problemen führen. Es treten Fehler wie Flimmern und Flackern auf oder es bleiben Artefakte zurück. Im Explorer werden viele Steuerelemente selbst gezeichnet, deshalb sind für jede, von *Control* abgeleitete, Klasse folgende Style Attribute definiert.

```
this.SetStyle(
  ControlStyles.AllPaintingInWmPaint |
  ControlStyles.UserPaint |
  ControlStyles.ResizeRedraw |
  ControlStyles.OptimizedDoubleBuffer, true);
```

Quellcode 5.1: Control Styles

Die Attribute *AllPaintingInWmPaint*, *UserPaint* und *OptimizedDoubleBuffer* sind für das Verhindern von Zeichenfehlern verantwortlich. Bei der Verwendung des *OptimizedDoubleBuffer* wird das Element zuerst in einem Buffer und dann erst auf dem Bildschirm gezeichnet. *AllPaintingInWmPaint* und *UserPaint* wird gesetzt, damit der Explorer selbst das Steuerelement zeichnen kann und nicht vom Betriebssystem gestört wird. Mit *ResizeRedraw* wird bei jeder Größenänderung eine Neuzeichnung des Elements angestoßen.

Darstellung von Text

Im Laufe der Entwicklung des Explorers stellte sich heraus, dass gelegentlich keine Schrift in den selbst gezeichneten Steuerelementen angezeigt wurde. Konkret trat dieser Fehler unter Windows 2000 auf. Das Problem war, dass die Texte nur mit der Methode *TextRenderer.DrawText* gezeichnet wurden. In der einfachsten Form verlangt diese Methode nur einen Punkt, bei dem der Text starten soll. Damit die Texte unter allen Versionen von Windows korrekt dargestellt werden können, muss der DrawText-Methode ein Rechteck mitgegeben werden, welches den zu zeichnenden Text umschließt. In Quelltext 5.2 ist der Code zu sehen, wie im Explorer Text auf selbst gezeichneten Elementen ausgegeben wird.

```
Size tmpSize = TextRenderer.MeasureText("TEXT", Font);
TextRenderer.DrawText(Graphics, "TEXT", Font, new Rectangle(0, 0,
   tmpSize.Width, tmpSize.Height), Color);
```

Quellcode 5.2: DrawText

Tastatureingaben

Da einzelne Funktionen des Explorers mit der Tastatur gesteuert werden, wurden einige Möglichkeiten getestet, wie die Eingaben am besten verarbeitet werden können. Eine Methode ist die Verarbeitung von Tastaureingaben, wenn in der Hauptform die Eigenschaft *KeyPreview* aktiviert wird. Dann übernimmt die Form die Abarbeitung der Ereignisse und die Steuerelemente innerhalb dieser Form können keine Tastaturereignisse mehr empfangen. Im Explorer wird diese Möglichkeit nicht verwendet, da die einzelnen Steuerelemente, neben den programmweiten Befehlen auch kontextbezogene Tastaturbefehle anbieten. Die Klasse *ExplorerRuntime* weist dem Graph, der Toolbar und dem Info-Bereich einen gemeinsamen Eventhandler zu. So kann zum Beispiel im Graph noch eine zusätzliche Abarbeitung, wie das Scrollen mit den Pfeiltasten, hinzugefügt werden. Bei der Implementierung ist darauf zu achten, dass eine von *Control* abgeleitete Klasse immer auch gewisse Tastenfunktionen erbt, wie zum Beispiel, das Verschieben einer Markierung zwischen einzelnen Steuerelementen mit den Pfeiltasten. Damit die Pfeiltasten mit eigenen Funktionen belegt werden können, ist das Überschreiben der Methode *IsInputKey* und das Herausfiltern der benötigten Tasten notwendig. Quelltext 5.3 zeigt, wie dies im Falle der Pfeiltasten funktioniert.

```
protected override bool IsInputKey(Keys keyData) {
  switch (keyData) {
    case Keys.Left:
    case Keys.Right:
    case Keys.Up:
    case Keys.Down:
      return true;
    }
  return base.IsInputKey(keyData);
}
```

Quellcode 5.3: Methode IsInputKey

5.2 Graph

Verwaltung der Knoten

Eine der Hauptaufgaben der Klasse Graph ist das Verwalten von Knoten. Durch das Monitoring der Runtime werden Funktionen für das Erzeugen und Entfernen von Knoten aufgerufen. Wie bereits beschrieben, gibt es Instanzknoten, Typknoten und ungebundene Typknoten. Jede dieser drei Arten wird in einer eigenen Liste verwaltet. Für die Instanzknoten wird ein Dictionary mit der zugehörigen ExtensionInfo als Schlüssel verwendet. Bei den ungebundenen Typknoten geschieht dies auf ähnliche Weise, nur mit der entsprechenden ExtensionTypeInfo als Schlüssel. Die registrierten ExtensionTypInfos werden in einem zweistufigen Dictionary verwaltet. Als Schlüssel wird in der ersten Ebene die SlotInfo und in der zweiten Ebene die PlugTypeInfo verwendet. Dies ist erforderlich, da aus Gründen der Übersicht zu jedem Slot bei dem ein Extension Type registriert ist, auch ein Knoten im Graph angezeigt wird.

```
Dictionary<ExtensionInfo, InstanceNode> instances;
Dictionary<SlotInfo, Dictionary<PlugTypeInfo, TypeNode>> types;
Dictionary<ExtensionTypeInfo, TypeNode> freeTypes;
```

Quellcode 5.4: Knotenlisten

Neben Knoten gibt es in einem Graph auch Kanten, diese werden aber nicht von der Klasse *Graph* selbst verwaltet. Das Speichern und Verwalten wird von der Layout-Extension übernommen. Sobald eine entsprechende Extension an den Explorer gepluggt wird, werden dieser Extension alle bisherigen Kanten übergeben, danach werden über die entsprechenden Funktionen des ILayout Interfaces die Listen der Layout-Extension aktuell gehalten. Der Codeabschnitt 5.5 aus der Klasse Graph zeigt, wie die Kanten aus der Plux.NET Runtime ausgelesen werden.

```
foreach (SlotInfo si in Runtime.Repository.SlotInfos) {
  foreach (PlugInfo pi in si.PluggedPlugInfos)
    AddInstanceConnection(si, pi);
  foreach (PlugTypeInfo pti in si.RegisteredPlugTypeInfos)
    AddTypeConnection(si, pti);
}
```

Quellcode 5.5: Ermitteln der Kanten

Zoommechanismus

Für große Graphen wird eine Funktion zum Zoomen benötigt. Anders als bei der Windows Presentation Foundation, basieren die Steuerelemente von Windows Forms nicht auf Vektoren, sondern sind in absoluten Größen angegeben. Das heißt, beim Zoomen muss die Größe und alle Abstände des Steuerelements neu berechnet werden, konkret wird die Größe der Knoten anhand der Schriftgröße berechnet. Die Klasse Graph stellt allen Steuerelementen diese Größe zu Verfügung und wenn sie neu gesetzt wird, werden alle Größen neu berechnet. Aus der Schriftgröße wird auch die Linenbreite für Einfassungen und Kanten berechnet. Von der Klasse Graph erhält die Layout-Extension die Schriftgröße als Zoomlevel und die Linienbreite. Diese zwei Variablen können benutzt werden, damit Layout des Graphs dem Zoomlevel entspricht. Die Funktionen *ZoomIn* und *ZoomOut* erhöhen beziehungsweise verringern den Schriftgrad um 1.0f. Mit der Funktion *ZoomNormal* wird der Schriftgrad wieder auf 8.25f gestellt.

Anordnen der Knoten

Der zentrale Punkt des Explorers ist der Extension-Graph. Das Anordnen seiner Knoten übernimmt die Layout-Extension. In dieser Arbeit wurde ein eigener Algorithmus zum Anordnen der Knoten entwickelt. Dieser Algorithmus kann einerseits Graphen mit mehreren Wurzeln anordnen und andererseits Teilgraphen mit einer definierten Wurzel anordnen. Graphen mit mehreren Wurzeln werden nebeneinander angeordnet. Zusätzlich kann der Graph um ein bestimmtes Offset verschoben werden. Der erste Schritt ist das hierarchische Sortieren der Knoten. In Quellcode 5.6 ist der Sortieralgorithmus zu sehen. Zu Beginn wird ein Dictionary mit der Wurzel als Schlüssel definiert. Zu jeder Wurzel gibt es wiederum ein Dictionary. Als Schlüssel wird der Level verwendet, die zugehörigen Daten sind eine Knotenliste. Nun wird mit der Funktion *findRoot* für jeden sichtbaren Knoten seine Wurzel und sein Level bestimmt. Die Variable *root* legt eine Wurzel für einen zusätzlich sortierten Teilgraph fest, wenn sie null ist, gibt es keine zusätzlichen Teilgraphen. Der Level ist der Abstand zur Wurzel und wird später für die Anordnung der Knoten benötigt.

```
Dictionary<INode, SortedDictionary<int, List<INode>>> sortedNodes
   = new Dictionary<INode, SortedDictionary<int, List<INode>>>();

foreach (INode node in allVisibleNodes) {
  KeyValuePair<INode, int> rootAndLevel = findRoot(node, 0, root);
  if (!sortedNodes.ContainsKey(rootAndLevel.Key))
    sortedNodes.Add(rootAndLevel.Key, new SortedDictionary<int,
       List<INode>>());
  if (!sortedNodes[rootAndLevel.Key].ContainsKey(rootAndLevel.
     Value))
    sortedNodes[rootAndLevel.Key].Add(rootAndLevel.Value, new List
       <INode>());
  sortedNodes[rootAndLevel.Key][rootAndLevel.Value].Add(node);
}
```

Quellcode 5.6: Sortieren der Knoten

Nach dem hierarchischen Sortieren der Knoten beginnt das Anordnen für jeden Teilgraph, dazu wird für jede Wurzel die Positionierung der Knoten durchgeführt. Die Idee des Algorithmus ist, dass die sortierten Knoten ihren Level entsprechend untereinander angeordnet werden. Zusätzlich sollen die Knoten auf der Y-Achse zentriert sein, es gibt sozusagen eine horizontale Mittellinie. Um die Knoten regelmäßig anordnen zu können, wird der größte Knoten jedes Levels benötigt. In der ersten Schleife wird dieser Wert berechnet und im Dictionary *maxXY* abgelegt. Zusätzlich wird auch die maximale Ausdehnung in X- und Y-Richtung des Levels ermittelt und in *maxX* sowie *maxX* gespeichert. Die Variablen *xOff* und *yOff* legen jeweils die Abstände zwischen den Knoten fest und werden bei den Berechnungen immer mit einbezogen. In der nächsten Schleife wird für jedes Level das Dictionary *startHeigh* befüllt. Dabei wird der Y-Wert eingetragen, bei dem der erste Knoten dieses Level platziert werden soll. Dies ist für die Zentriertheit auf der Y-Achse notwendig. In der letzen Schleife wird nun jedem Knoten seine endgültige Position zugewiesen. Diese Position hängt von einem globalen Offset und den vorangegangenen Knoten ab. Da der Algorithmus mehrere Wurzeln verwalten kann, wird zum Schluss noch ein Offset für die nächste Wurzel berechnet. Der nächste Teilgraph wird rechts vom gerade bearbeiteten positioniert.

```
int xOffSet = 0;

foreach (SortedDictionary<int, List<INode>> list in sortedNodes.
   Values) {
  Dictionary<int, Size> maxXY = new Dictionary<int, Size>();
  Dictionary<int, int> startHeight = new Dictionary<int, int>();

  int maxX = 0, maxY = 0;
  int xOff = zoomLevel * 6, yOff = zoomLevel * 4;
  foreach (int level in list.Keys) {
    list[level].Sort();
    maxXY.Add(level, new Size(0, 0));
    foreach (INode node in list[level]) {
      maxXY[level] = new Size(maxXY[level].Width > node.Size.Width
          + xOff ? maxXY[level].Width : node.Size.Width + xOff,
         maxXY[level].Height + node.Size.Height + yOff);
    }
    maxXY[level] = new Size(maxXY[level].Width, maxXY[level].
       Height - yOff);
    maxX += maxXY[level].Width;
    maxY = maxY>maxXY[level].Height ? maxY : maxXY[level].Height;
  }

  foreach (int level in list.Keys) {
    startHeight.Add(level, (maxY/2) - (maxXY[level].Height/2));
  }

  int tmpWidth = 0;
  foreach (int level in list.Keys) {
    foreach (INode node in list[level]){
      node.Location = new Point(tmpWidth + xOffSet + offset.X,
         startHeight[level] + offset.Y);
      startHeight[level] += node.Size.Height + yOff;
    }
    tmpWidth += maxXY[level].Width + xOff;
  }

  xOffSet += xOff + tmpWidth;
}
```

Quellcode 5.7: Anordnen der Knoten

Zeichnen der Kanten

Alle Kanten werden bei der Neuzeichnung des Graphs in der Klasse Graph gezeichnet. Sie werden als Pfeile dargestellt, beginnen bei einem Plug und enden bei einem Slot. Plug und Slot stellen dafür über ein Interface einen Punkt zu Verfügung. Die eigentliche Funktion wird von der Layout-Extension implementiert. Wenn ein Plug selektiert ist, wird zur Kennzeichnung am Beginn des Pfeils ein kleines Rechteck gezeichnet. Darauf folgt das Zeichnen der Linie. Diese Linie ist eine Bezierkurve mit Stützpunkten kurz nach dem

Slot und kurz vor dem Plug. Quellcode 5.8 zeigt, wie in der Layout-Extension die Kanten zwischen den einzelnen Instanzknoten gezeichnet werden.

```
Pen instancePen = new Pen(ArrowColor, lineWidth);
foreach (Edge edge in instanceEdges) {
  if (edge.ToPlug.Selected) {
    int height = edge.ToPlug.Height / 2;
    if (height == 0) height = 1;
    Grahics.FillRectangle(new SolidBrush(ArrowColor), edge.ToPlug.
       AttachPoint.X - lineWidth * 3, edge.ToPlug.AttachPoint.Y -
       height, lineWidth * 3, height * 2);
  }
  Grahics.DrawBezier(instancePen, edge.FromSlot.AttachPoint, new
     Point(edge.FromSlot.AttachPoint.X + zoomLevel * 4, edge.
     FromSlot.AttachPoint.Y), new Point(edge.ToPlug.AttachPoint.X
     - zoomLevel * 4, edge.ToPlug.AttachPoint.Y), edge.ToPlug.
     AttachPoint);
}
```

Quellcode 5.8: Zeichnen der Kanten

Wenn zusätzlich auch Typknoten angeordnet werden, wird analog vorgegangen. Markierungen werden nicht benötigt, aber damit eine Linie gezeichnet wird, müssen Beginn- und Endknoten sichtbar sein.

5.3 Knoten

Knotengröße und Ansicht

Die Knotengröße wird mehrstufig berechnet und hängt direkt vom Zoomlevel ab, deshalb haben die Klassen *BaseNode*, *BaseSlot*, *BasePlug* und *ToggleButton* auch Zugriff auf das Feld *Font* der Klasse *Graph*. Auch die Einstellungen der angepassten Ansicht werden für die Größenberechnung benötigt. Die Berechnung der Größe startet in der *BaseNode* und zu Beginn werden von ihren *BaseSlots*, *BasePlugs* und *ToggleButtons* die Größen berechnen. Ein *BasePlug* ist normalerweise so groß, wie der Name des zugehörigen Plugs. Wenn die Parameter angezeigt werden, dann kommt pro Parameter eine Zeile hinzu. Auch ein *BaseSlot* ist nur so groß wie der Name des Slots. Durch Anzeige von Parametern und Togglebuttons verändert sich die Größe. Bei den Togglebuttons gibt es immer nur eine Größe. Nachdem alle Größen bekannt sind, berechnet die *BaseNode* eine Referenzgröße aus dem Zoomlevel und dem Namen des Knotens. Diese Größe ist sehr wichtig, da sie für alle Abstands- und Größenberechnungen herangezogen wird. Die Größe der *BaseNode* berechnet sich schlussendlich dadurch, wie groß die Bereiche für *BasePlugs* und *BaseSlots* sind, beziehungsweise ob diese überhaupt angezeigt werden sollen. Nach der Größenberechnung wird durch die *CalculateSize*-Funktion die Größe der *BaseNode* gesetzt und zusätzlich werden auch noch Punkte berechnet, die angeben, wo die einzelnen Steuerelemente innerhalb der *BaseNode* platziert werden sollen. Die eigentliche Platzierung der Steuerelemente passiert beim Neuzeichnen des Steuerelements. Werden keine Slots oder Plugs angezeigt, so werden diese beim Neuzeichnen unsichtbar gesetzt.

Aufbau eines dynamischen Menüs

Eine Hauptfunktion des Explorers sind die dynamischen Kontextmenüs, denn sie stellen sicher, dass immer die aktuellsten Menüeinträge zu Verfügung stehen. Für jede Plux.NET Komponente gibt es ein Steuerelement, bei dem es wiederum ein Kontextmenü gibt. Da der Aufbau der Menüs immer nach dem gleichen Schema abläuft, wird er anhand des Steuerelements *InstancePlug* für PlugInfos demonstriert. Die Funktion *PlugContextMenuOpening* aus Quellcode 5.9 wird an das *Opening*-Event des Kontextmenüs gehängt, damit das Menü bei jedem Aufruf neu aufgebaut werden kann. Als erstes wird das Kontextmenü gelöscht und eine Liste angelegt, welche alle unpluggbaren Slots enthalten wird. Diese Liste wird mit den aktuell gepluggten Slots aufgefüllt und sortiert. Nun wird der Menüeintrag erstellt, falls es keine Slots zum unpluggen gibt, so wird er deaktiviert dargestellt. Wenn es aber Einträge gibt, dann wird als erstes ein Menüeintrag hinzugefügt, mit dem alle Slots unpluggt werden können und danach eine *Unplug*-Aktion für jeden einzelnen. Als *Tag* wird immer der Slot an den Einträg hinzugefügt, für den die *Unplug*-Aktion ausgeführt werden soll. Dieser Vorgang wird bei allen anderen Kontextmenüs genau so wiederholt, einzig dass Füllen der Listen unterscheidet sich.

```
void PlugContextMenuOpening(object s, CancelEventArgs e) {
  ToolStripMenuItem item, subItem;
  plugContextMenu.Items.Clear();
  List<SlotInfo> canUnplug = new List<SlotInfo>();
  canUnplug.AddRange(plug.PluggedInSlots);
  canPlug.Sort(new SlotInfoComparer());

  item = new ToolStripMenuItem("Unplug");
  if (canUnplug.Count > 0) {
    item.DropDownItems.Add(new ToolStripMenuItem("All", null,
       delegate { plug.Unplug(); }));
    item.DropDownItems.Add("-");
    foreach (SlotInfo si in canUnplug) {
      subItem = new ToolStripMenuItem(si.Name + " (" + si.
         ExtensionInfo.Name + ")", null, Unplug);
      subItem.Tag = si;
      item.DropDownItems.Add(subItem);
    }
  } else {
    item.Enabled = false;
  }
  plugContextMenu.Items.Add(item);
}

void Unplug(object sender, EventArgs e) {
  ToolStripMenuItem item = (ToolStripMenuItem)sender;
  plug.Unplug((SlotInfo)item.Tag);
}
```

Quellcode 5.9: Aufbau Kontextmenü

Erstellen von neuen Extensions

Sowohl im Graph, als auch in der Toolbar kann von einem Extension Type eine neue Extension erzeugt werden. Dieses Erzeugen hat zur Folge, dass ein neuer Knoten zum Graph hinzugefügt wird. Normalerweise würde der Layout-Algorithmus den neuen Knoten im Graph platzieren und er müsste gesucht werden. Es gibt aber eine Funktion, dass der erzeugte Knoten immer beim Erzeuger platziert wird und damit direkt mit dem neuen Knoten weitergearbeitet werden kann. Wenn im Kontextmenü des Typknoten eine neue Extension erzeugt wird, wird diese dem Feld *CreatedExtension* zugewiesen. Dieser Setter ist in Quelltext 5.10 zu sehen. Falls eine Shared-Extension erzeugt wurde, wird diese vom Extension Type geholt. Danach wird der entsprechende Instanzknoten vom Graph gesucht, wenn dieser existiert, wird seine Position geändert und der Knoten wird markiert.

```
public ExtensionInfo CreatedExtension {
  set {
    createdExtension = value;
    if (createdExtension == null) {
      createdExtension = type.GetSharedExtension();
    }
    if (createdExtension != null) {
      InstanceNode instance = graphPanel.FindInstanceNode(
         createdExtension);
      if (instance != null) {
        instance.Location = new Point(Location.X + 15 * graphPanel
           .LineWidth, Location.Y + 15 * graphPanel.LineWidth);
        graphPanel.Invalidate();
        instance.Focus();
} } } }
```

Quellcode 5.10: Setter CreatedExtension

Togglebuttons

Die Togglebuttons sind Schalter für die bool'schen Eigenschaften der Plux.NET Komponenten. So können zum Beispiel für eine SlotInfo die Eigenschaften AutoRegister, AutoOpen, AutoPlug, LazyLoad und Unique umgeschalten werden. Die Togglebuttons sind halbautomatisch, das heißt nach einem Mausklick ändern sie ihren Zustand und ihr Aussehen erst dann, wenn sie von der Runtime ein Event empfangen. Zusätzlich ändern sie ihren Zustand auch, wenn zum Beispiel die Konsole die Eigenschaften ändert. Um sich selbständig ändern zu können, sind der Attributname (*attributeNameLong*) und die Komponente (*attributeObject*) notwendig. Jeder Togglebutton hängt an das *PropertyChanged*-Event seiner Komponente den Listener aus Quellcode 5.11. Sobald sich nun eine Eigenschaft einer Komponente ändert, kann jeder Togglebutton feststellen, ob es ihn betrifft und sich gegebenenfalls neu zeichnen.

```
void AttributeChanged(object a, PropertyChangedEventArgs e) {
  if (sender.Equals(attributeObject) && e.PropertyName.Equals(
     attributeNameLong)) {
    Refresh();
} }
```

Quellcode 5.11: PropertyChanged Funktion

Kapitel 6

Fazit und Ausblick

Die Entwicklung des Explorers war eine herausfordernde Aufgabe. Vor allem die Windows Forms Programmierung war nicht immer einfach, da einige Schwächen dieses Frameworks aufkamen, wenn nicht nur die vorgegebenen Steuerelemente verwendet werden. So mussten einige Ideen für die Benutzeroberfläche angepasst werden, damit sie trotzdem implementiert werden konnten. Auch die Arbeit mit Plux.NET war höchst interessant, vor allem da die Programmierung der Bachelorarbeit parallel zur Entwicklung von Plux.NET statt fand. Durch diese Parallelität gewann ich Einblick in die Entwicklung eines komplett neuen Plugin-Frameworks.
Der Explorer dieser Bachelorarbeit ist voll funktionsfähig, doch es gibt immer Raum für Änderungen und Verbesserungen. Wie die Zukunft des Explorers aussehen könnte, ist nachfolgend aufgeführt.

- Da die Implementierung des Layout-Algorithmus für den Graph einfach gehalten ist, ist hier Verbesserungspotential vorhanden. Durch ein Austauschen der bisherigen Layout-Extension, kann zum Beispiel mit Hilfe von GraphViz ein besseres Ergebnis erzielt werden. GraphViz ist eine OpenSource-Software zum Visualisieren von Graphen [8].

- Eine Implementierung mit der Windows Presentation Foundation würde mehr Freiraum in der Gestaltung der Benutzeroberfläche zulassen als die bisherige Windows Forms 2.0 Implementierung.

- Der Explorer könnte noch mehr die Ideen von Plux.NET verwenden und so noch modularer werden. Zum Beispiel könnten die Knoten auch als Erweiterung implementiert werden und so zur Laufzeit ausgetauscht werden.

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] Aufgabenstellung der Bachelorarbeit, http://ssw.uni-linz.ac.at/Teaching/Projects/PluginExplorer/Aufgabenstellung.PluginExplorer.pdf, aufgerufen am 10.01.2010

[2] Reinhard Wolfinger: *Dynamic Composition with Plux.NET: Composition Model, Composition Infrastructure*, PhD Thesis, Johannes Kepler University, Linz, Austria, 2010.

[3] Chris Sells, Michael Weinhardt: *Windows Forms 2.0 Programming*, Addison Wesley Professional, 2006.

[4] Iulian Serban, Dragos Brezoi, Tiberiu Radu, Adam Ward: *GDI+ Custom Controls with Visual C# 2005*, Packt Publishing, 2006.

[5] Matthew MacDonald: *Pro .NET 2.0 Windows Forms and Custom Controls in C#*, APress, 2006.

[6] Simon Robinson, Christian Nagel, Jay Glynn, Morgan Skinner, Karli Watson, Bill Evjen: *Professional C#*, Wiley Publishing, 2004.

[7] Erik Brown: *Windows Forms in Action*, Manning, 2006.

[8] Projektseite von GraphViz, http://www.graphviz.org/, aufgerufen am 10.01.2010