UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

Design und Entwicklung einer schnellen Laufzeitumgebung für CrypTool 2.0

Nils Kopal
Matrikelnummer: 2233615

Abteilung Informatik und angewandte Kognitionswissenschaft
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

28. September 2010

Prüfer:
Prof. Dr. Ing. Torben Weis
Prof. Dr. rer. nat. Pedro José Marrón
Betreuer:
Dr. Arno Wacker
Dipl.-Inform. Matthäus Wander

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

CrypTool ist ein Programm, welches der Benutzerzielgruppe (Lehrern, Schülern, Dozenten, Studenten sowie kryptographisch Interessierten) eine Lehr- und Lernplattform für kryptographische und kryptoanalytische Algorithmen und Verfahren bietet. Die Entwicklung ist ein Open-Source Projekt, an dem sich unter anderem die Universität Duisburg-Essen, die Universität Siegen, die Deutsche Bank und viele weitere Institutionen sowie auch einzelne Entwickler beteiligen. Seit 2007 wird an CrypTool 2.0 (CT2) gearbeitet, welches der Nachfolger von CrypTool ist. In dieser Bachelorarbeit wurde für CT2 das Modell für einen neuen Editor entwickelt, der den bisherigen Editor ersetzt und einige Neuerungen bietet, die im Laufe dieser Arbeit beschrieben sind. Teil dieses neuen Editors ist eine neue Laufzeitumgebung, die als Schwerpunkt innerhalb dieser Arbeit entwickelt wurde.

1.1 Motivation

Technisch wurde das erste CrypTool mit Hilfe der Microsoft Foundation Classes (MFC) in der Sprache C++ entwickelt. Als CrypTool das für ein Softwareprodukt übliche Lebenszyklusende erreicht hatte, wurde vom Entwicklerteam die Entscheidung für die Neuentwicklung eines Nachfolgers getroffen. Für den Nachfolger CT2 fiel die Wahl auf eine pluginbasierte Architektur in der Sprache C#. Vorteile durch die neue Architektur waren die Möglichkeit vom verteilten Entwickeln mit standardisierten Schnittstellen zwischen den Plugins und eine einfach zu verstehende Architektur, die einen schnellen Einstieg in die Entwicklung für Neueinsteiger bietet. CT2 ist im Gegensatz zu CrypTool "grafisch". Es bietet dem Benutzer die Möglichkeit diverse kryptographische Algorithmen als Plugins auf den sogenannten Workspace mittels "Drag and Drop" zu ziehen und diese dann miteinander zu verbinden. Hat der Benutzer alle notwendigen Ein- und Ausgänge der Algorithmen verbunden, kann er mittels eines Play-Buttons die so gebildeten Ketten ausführen. Die Ergebnisse der einzelnen Algorithmen werden dann grafisch und/oder textuell innerhalb der Anzeigen der einzelnen Plugins dargestellt. Das Zusammen-klicken der Pluginketten wird durch ein weiteres spezielles Plugin, den sogenannten Editor, ermöglicht. Der bisherige Editor wurde von Thomas Schmidt im Rahmen seiner Diplomarbeit [Sch08] entwickelt. Der unter Usability-Gesichtspunkten hervorragend entwickelte Editor zeigt jedoch mittlerweile einige Schwächen in der deterministischen Ausführung der Plugins und ist aufgrund seines Designs nicht auf Geschwindigkeit ausgelegt. Wegen seiner monolithischen Architektur wurde daher eine Neuentwicklung des Editors, basierend auf dem Model-View-Controller-Pattern (MVC) beschlossen. Die Entwicklung des neuen Models, das sich hinter dem neuen Editor verbirgt, wurde im Rahmen dieser Bachelorarbeit durchgeführt. Die Entwicklung der Oberfläche (View und Controller) wurde von Viktor Matkovic in seiner Bachelorarbeit [Mat10] durchgeführt.

1.2 Problembeschreibung

Die Entwicklung eines neuen Editors für CT2 ist aus vielerlei Gründen notwendig geworden. Zum einen wäre da die dem alten Editor zugrunde liegende Architektur. Es herrscht keine klare Model-View-Controller Struktur, die in der Entwicklung von modernen Oberflächen de facto Standard ist. Dies führt dazu, dass Änderungen sehr schwer in den alten Editor eingebracht werden können und neue Entwickler den Editor beziehungsweise seinen Aufbau kaum oder gar nicht verstehen. Das nächste Problem ist die Ausführung der im Editor gezeichneten Pluginketten. Die Ausführungsgeschwindigkeit der Pluginketten im alten Editor ist langsam und selbst einfache Pluginketten dauern eine gewisse Zeit, bis diese ausgeführt wurden. Des weiteren kommt es in komplizierten Pluginketten zu sogenannten Race-Conditions, da die Weitergabe von Daten von einem zum anderen Plugin (über mehrere Threads hinweg) bisweilen schief geht. So kann es vorkommen, dass das eine oder andere Plugin mit unvollständigen Daten oder sogar mehrfach ausgeführt wird. Dies rührt daher, da der alte Editor keine Flusskontrolle zwischen der Ausführung der einzelnen Plugins bietet. Ein anderer Punkt war der Wunsch einer “moderner aussehenden Oberfläche” des Editors. Die Entwicklung dieser Oberfläche wurde von Viktor Matkovic in seiner Bachelorarbeit durchgeführt. Die Speicherung der Pluginketten im alten Editor wurde durch einfache Objektserialisierung vorgenommen, welche ein nicht Menschen-lesbares Binärformat erzeugt.

1.3 Aufgabenstellung

Ausgehend von den, in der Problembeschreibung beschriebenen Problemen, lassen sich folgende Anforderungen für die Entwicklung und Konzeption des neuen Editors nennen:

- (R01) Die Nutzung des Model-View-Controller Musters dient als Grundlage des Designs des neuen Editors
- (R02) Eine höhere Ausführungsgeschwindigkeit im Gegensatz zum alten Editor soll erreicht werden
- (R03) Eine deterministische Ausführung der Pluginketten ohne Race Conditions soll erreicht werden
- (R04) Es soll eine Flusskontrolle während der Ausführung der Plugins stattfinden
- (R05) Ein sauber strukturiertes, an die Eigenschaften des Editors angepasstes, Datenformat für die Speicherung des Models soll umgesetzt werden
- (R06) Der neue Editor muss alle Plugins genau so ausführen können, wie es der alte Editor bereits kann

Auf Grundlage dieser sechs Anforderungen wurde die Entwicklung des neuen Editors vorgenommen.

1.4 Aufbau der Arbeit

Im folgenden Kapitel 2 werden die Grundlagen vorgestellt, die bei der Entwicklung des neuen Editors und der neuen Ausführungsmaschine verwendet wurden.

Danach wird in Kapitel 3 gezeigt, wie mit Hilfe von Petrinetzen das Konzept für das neue Model und die neue Ausführungsmaschine entwickelt wurde. Hier wird auch die Persistierung des Models, also das Laden und Speichern beschrieben.

Die Implementierung des entwickelten Konzepts folgt in Kapitel 4.

Nachdem das Konzept sowie die Implementierung des neuen Models und der Ausführungsmaschine gezeigt wurden, wird die entwickelte Implementierung in Kapitel 5 bezüglich der Ausführungsgeschwindigkeit und der Speichernutzung analysiert. In diesem Kapitel wird auch die Kompatibilität des neuen zum alten Editor dargestellt.

Im letzten Kapitel 6 wird ein Ausblick für mögliche Weiterentwicklungen des neuen Editors gegeben. Am Ende dieser Arbeit steht eine Zusammenfassung sowie ein Fazit.

2 Grundlagen

Dieses Kapitel beschreibt die technologischen Grundlagen, die für die Konzeption und Implementierung des neuen CT2 Editors genutzt wurden. Zunächst wird CT2 für sich betrachtet und sein Aufbau und seine Funktionen erklärt. Hier werden auch grundlegende Funktionen aufgelistet, die jeder CT2 Editor bieten muss. Danach wird das Model-View-Controller-Muster beschrieben, das als Architekturgrundlage des neuen Editors dient. Daraufhin werden Petri-Netze kurz erläutert, die als Grundlage für die Entwicklung einer neuen deterministischen Ausführungsmaschine genutzt wurden. Das vorletzte Unterkapitel bildet eine Beschreibung des Gears4Net-Frameworks, das die Entwicklung von massiv parallelen Anwendungen ermöglicht. Gears4Net dient als Grundlage der Ausführungsmaschine. Als letztes Unterkapitel wird die Auszeichnungssprache XML erläutert, da diese für die Persistierung des neuen Models genutzt wurde.

2.1 CrypTool 2.0

CT2 ist eine, auf WPF-Komponenten [JW08] basierte, Windows-Anwendung, die fast vollständig mit Hilfe von .Net-Technologien in der Programmiersprache C# [JL08] entwickelt wurde. CT2 bietet eine Oberfläche, die im Folgenden kurz beschrieben wird.

2.1.1 Die Oberfläche

Wie aus den Abbildungen 2.1 und 2.2 ersichtlich, besteht die eigentliche CT2-Oberfläche aus fünf GUI-Elementen. Zum einen gibt es die Toolbar, die z. B. das Starten und Stoppen von Pluginketten, deren Speichern, Laden und so weiter ermöglicht. Dann gibt es eine weitere Leiste auf der linken Seite, die alle verfügbaren Plugins darstellt. Auf der rechten Seite kann man Einstellungen zu selektierten Plugins vornehmen. Am unteren Rand gibt es ein Meldungsfenster, indem der Benutzer Meldungen der Plugins einsehen kann. Das wichtigste GUI-Element ist aber der sogenannte Workspace. Er dient als Arbeitsfläche, auf der die Pluginketten erstellt werden können. Der Workspace liegt in der Mitte der Oberfläche von CT2. Die Anordnung der einzelnen Elemente ist variabel, da sie nach Belieben ein- oder ausgeblendet werden können und in ihrer Position auch an anderen Stellen angedockt werden können.

Das für die Entwicklung wichtigste grafische Element von CT2 ist der Workspace. Dieser entspricht der weißen Fläche in Abbildung 2.2, beziehungsweise deren Entsprechung in Abbildung 2.1. Der Workspace bietet "Platz" für den Editor, mit dem die Pluginketten zusammen-geklickt werden können. Um den Startvorgang des Editors und dessen Auftauchen im Workspace zu verstehen, muss man sich den Startvorgang von CT2 genauer anschauen. Das erste, was CT2 anzeigt, sobald es gestartet wird, ist der sogenannte Splash-Screen. Neben dem CT2-Logo und Lizenzinformationen zeigt der Splash-Screen Statusinformationen bezüglich des Ladens aller Plugins, die im CT2-Plugins-Verzeichnis

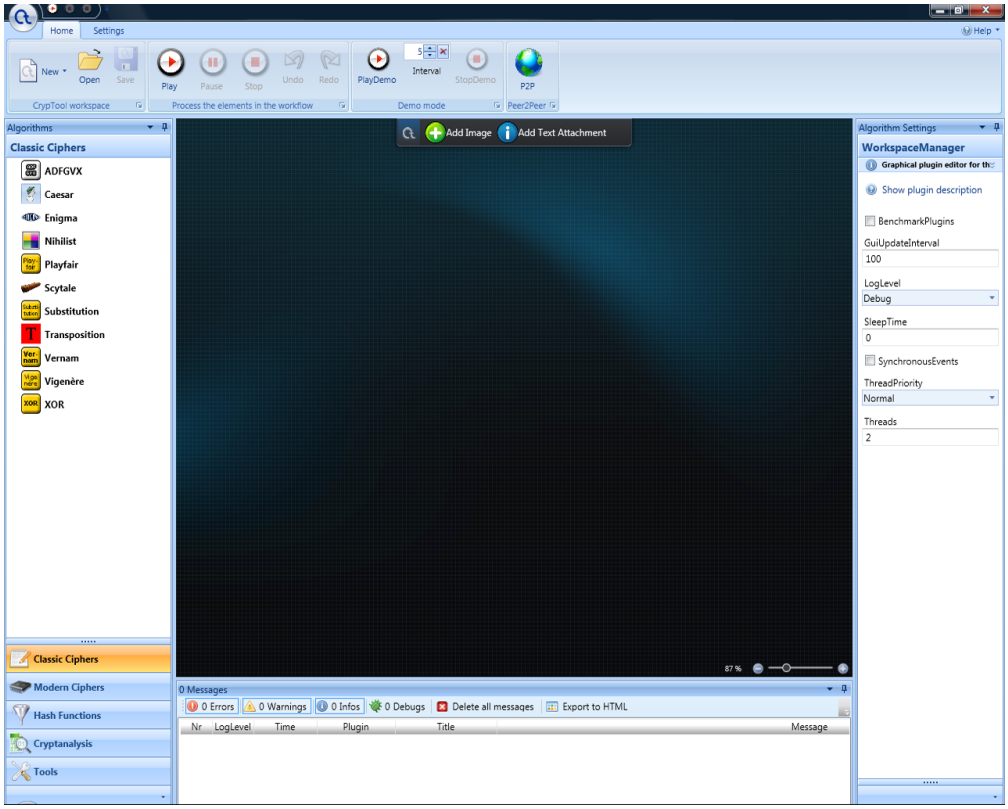

Abbildung 2.1: Screenshot der CT2-Oberfläche

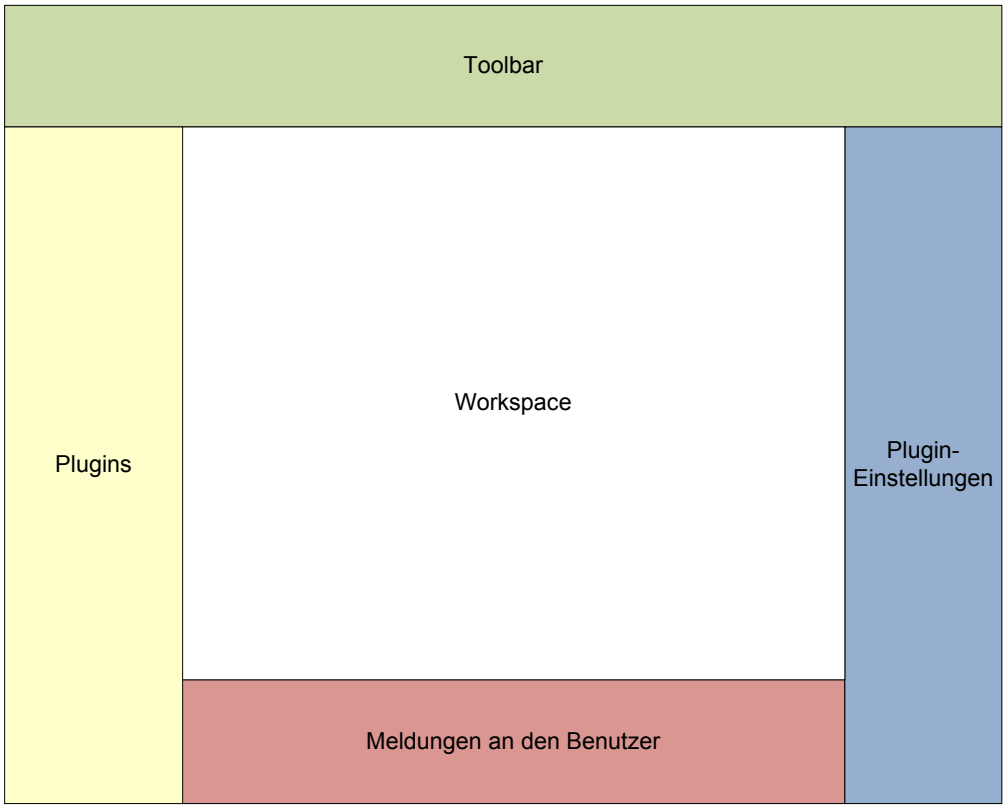

Abbildung 2.2: Schematischer Aufbau der CT2-Oberfläche

enthalten sind. Ein Plugin zeichnet sich dadurch aus, dass es eine Reihe von Ein- und Ausgängen besitzt, sowie eine Position und Größe auf dem Workspace besitzt. Ein Plugin wird in eine sogenannte .Net-Assembly verpackt und, wie bereits erwähnt, in ein spezielles Plugins-Verzeichnis gelegt. CT2 bietet alle geladenen Plugins in der linken Plugins-Übersicht an. Mittels "drag & drop" können dann Plugins auf den Workspace gezogen werden und werden dort mittels eines viereckigen Symbols dargestellt. Ein CT2-Editor kann bestimmte Events und Eingaben verarbeiten. So muss ein Editor Pluginketten, in einem eigenen Format, laden und speichern können. Er muss auf Start- und Stop-Events von der CT2-Oberfläche reagieren können und eine Oberfläche anbieten, die innerhalb des Workspaces dargestellt werden kann.

2.1.2 Die Plugins

Wie bereits bei der Beschreibung der Oberfläche erwähnt, ist ein Plugin ein, in eine .Net-Assembly "verpacktes" Programm. Es wird von CT2 automatisch geladen, sofern sich seine Assembly innerhalb des Plugin-Verzeichnisses von CT2 befindet. Diese Plugins können mit Hilfe der Maus aus der Pluginleiste auf den Workspace gezogen werden.

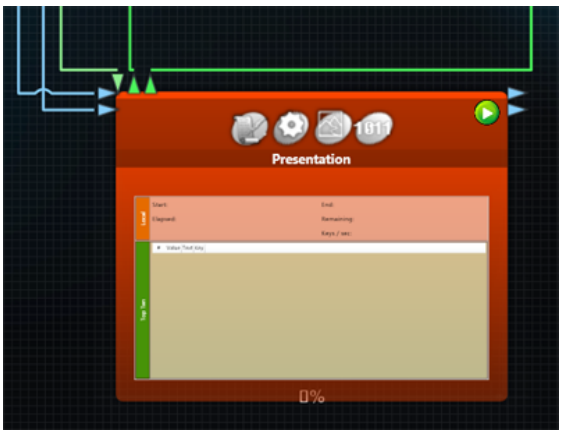

Abbildung 2.3: Screenshot eines CT2-Plugins innerhalb des neuen Editors

CT2-Plugins kapseln kryptographische, kryptoanalytische oder jedes andere deterministisch berechenbare Verfahren. Dies kann z. B. ein Plugin, das den RSA-Algorithmus beinhaltet, aber auch ein Plugin, das z. B. Texteingabe ermöglicht, sein. Ein Plugin zeichnet sich dadurch aus, dass es mehrere Ein- und Ausgänge besitzt. Ein Ein- oder Ausgang kann als optional markiert sein, dann ist es nicht zwingend notwendig, diesen mit einer Datenleitung zu verbinden. Plugin Ein- und Ausgänge werden mittels des Editors miteinander verknüpft um Pluginketten zu erstellen. Plugins innerhalb dieser Ketten können auch vollvermascht und nicht nur linear miteinander verbunden sein. Um Plugins miteinander zu verknüpfen, müssen die Ein- und Ausgänge vom Editor grafisch dargestellt werden. Dies geschieht z. B. durch sogenannte Konnektoren, die im alten Editor als Dreiecke am Rand eines Plugins dargestellt wurden. Abbildung 2.3 zeigt beispielhaft die von Viktor Matkovic entwickelte neue grafische Repräsentation eines Plugins. Es besitzt dreieckige Konnektoren und angeschlossene Verbindungen zu anderen Plugins. Konnektoren kann der Benutzer dann mit Hilfe der Maus durch eine Linie

miteinander verbinden. So ist der Ausgang des einen mit dem Eingang des anderen Plugins verknüpft. Es existieren auch Plugins, die nur Ausgänge besitzen. Diese sind dann startfähig und können, sobald sie in einer Pluginkette untergebracht sind, durch Starten der Ausführung gestartet werden. Ein Beispiel ist ein Texteingabeplugin, das dem Benutzer eine Schaltfläche bietet, um Daten einzugeben. Genauso gibt es das Gegenstück dazu, um Text auszugeben. Dieses Plugin ist ein Beispiel für Plugins, die nur Eingänge besitzen und das Ende einer Kette bedeuten können. Plugins besitzen viele unterschiedliche Funktionen, wobei die wichtigste die Ausführungsfunktion ist. Diese wird genau dann ausgeführt, wenn das Plugin durch den Editor gestartet wird. Ein Plugin kann auch eine grafische Repräsentation besitzen, die in Form einer WPF-Komponente realisiert wurde. Dies wird als "Presentation" bezeichnet. Editoren sollten diese grafische Repräsentation eines Plugins in ihrer eigenen grafischen Repräsentation innerhalb des Workspace darstellen. Damit Plugins konfiguriert werden können, können für jedes Plugin Einstellungen verändert werden. Einstellungen eines Plugins werden innerhalb von CT2 an der rechten Seite angezeigt (siehe Abbildungen 2.1 und 2.2), sobald der Editor meldet, dass ein Plugin selektiert wurde.

2.1.3 Grundlegende Funktionen, die ein CT2-Editor bieten muss

Ausgehende von den Beschreibungen zu CT2 und den Plugins, lassen sich folgende grundlegende Funktionen auflisten, die ein CT2-Editor bieten muss:

- (F01) Ein CT2-Editor muss eine grafische Repräsentation besitzen, die innerhalb des CT2 Workspace dargestellt werden kann
- (F02) Das Hinzufügen von neuen Plugins auf den CT2 Workspace muss möglich sein
- (F03) Das Bewegen, Neupositionieren und Löschen von Plugins innerhalb des Editors muss möglich sein
- (F04) Das Verknüpfen von Ein- und Ausgängen von Plugins muss innerhalb des Editors möglich sein
- (F05) Besitzt ein Plugin eine eigene grafische Repräsentation, dann muss diese innerhalb der grafischen Repräsentation des Editors durch den Benutzer eingesehen werden können
- (F06) Besitzt ein Plugin Konfigurationsmöglichkeiten, so muss der Editor ermöglichen, diese durch den Benutzer einzusehen und Veränderungen an diesen vorzunehmen
- (F07) Die mittels des Editors gebildeten Pluginketten müssen ausgeführt und die Ausführung muss gestoppt werden können
- (F08) Das Laden, Speichern und Neuerstellen einer Pluginkette muss möglich sein

Hierunter fällt auch, dass sowohl die Plugins an sich, als auch deren Ein- und Ausgänge und die Verbindungen zwischen diesen eine grafische Repräsentation erhalten müssen, mit denen der Benutzer interagieren kann.

2.2 Model-View-Controller

Das **Model-View-Controller-Muster** (kurz **MVC-Muster**) [Fow02] ist ein Strukturmodell für die Entwicklung von Anwendungen mit Benutzeroberfläche(n) (die **View**(s)), welche ein Datenmodell (das **Model**) anzeigt und manipulierbar macht. Der Benutzer bekommt das Datenmodell in Form einer oder mehrerer Views präsentiert und kann dieses mit deren Hilfe manipulieren. Die Manipulationen erfolgen durch Benutzereingaben, die durch den **Controller**, welcher die dritte Komponente des MVC darstellt, verarbeitet werden. Durch die Loslösung des Models von der Anzeigelogik und der Verarbeitung der Benutzereingaben, können theoretisch alle Teile einzeln für sich ausgetauscht werden. Abbildung 2.4 dient zur Veranschaulichung der Zusammenhänge zwischen Model, View und Controller.

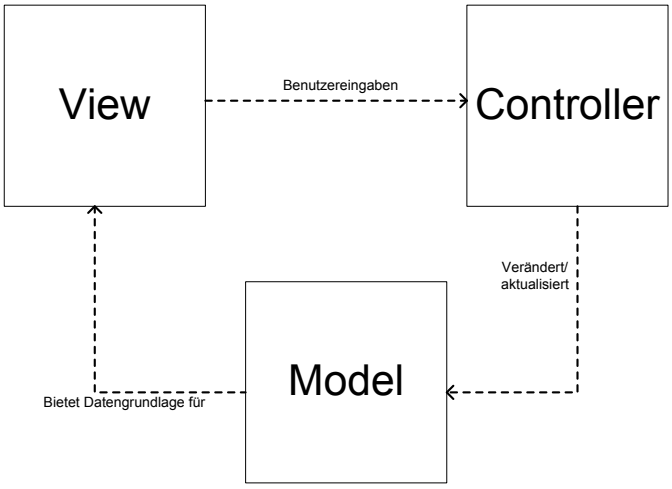

Abbildung 2.4: Model-View-Controller-Muster

Häufig wird das MVC-Muster durch ein weiteres Muster erweitert, dem sogenannten **Observer-Muster**. Beim Observer-Muster kann sich beim Datenmodell ein sogenannter **Beobachter** registrieren. Wird das Datenmodell verändert, ruft es automatisch bei allen registrierten Beobachtern eine festgelegte Methode (z. B. update()) auf und informiert diesen so von der Veränderung. Die Beobachter wiederum können nun auf diese Veränderung ihres Datenmodells reagieren. Beim MVC-Muster registrieren sich die Views als Beobachter bei ihrem Model. So wird gewährleistet, dass es zu keiner Inkonsistenz zwischen den auf der View dargestellten Informationen und dem im Hintergrund liegenden Daten kommen kann.

2.3 Petri-Netze

Ein **Petri-Netz** [LP08] ist ein Modell zur Darstellung und Analyse von nebenläufigen Systemen. Es besteht aus zwei unterschiedlichen Arten von Knoten, den **Stellen** und den **Transitionen**, die mittels gerichteter Kanten miteinander verbunden werden. Stellen werden nur mit Transitionen verbunden und Transitionen nur mit Stellen. Es gibt keine direkte Verbindung zwischen zwei Stellen oder zwei Transitionen. Eine Stelle

kann eine bestimmte Anzahl von sogenannten Tokens aufnehmen (ihre Kapazität). Eine Transition entnimmt allen Stellen, die mit einer Kante auf sie zeigen, eine bestimmte Anzahl von Tokens und erhöht alle Tokens der Stellen, auf die sie zeigt ebenfalls mit einer bestimmten Anzahl von Tokens. Dies geschieht genau dann, wenn alle Eingangsstellen der Transition über ausreichend Tokens verfügen und alle Ausgangsstellen über genügend Aufnahmekapazität für neue Tokens verfügen. Dieser Vorgang wird auch Schaltung genannt. Existieren mehrere schalt-bereite Transitionen, so schalten diese unabhängig voneinander in einer willkürlichen Reihenfolge parallel. Petri-Netze verfügen bei ihrer Instanziierung über eine festgelegte Anzahl von Tokens, der sogenannten Startmarkierung. Verfügt ein Petri-Netz über keine schalt-bereite Transition mehr, so wird es als tot bezeichnet.

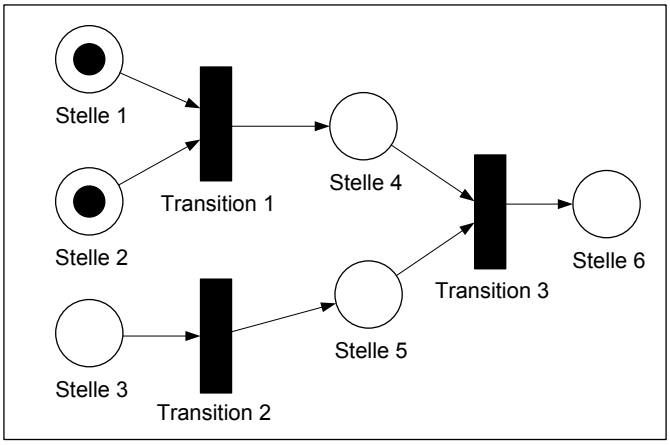

Abbildung 2.5: Beispiel für ein Petrinetz

Es existiert eine einfache grafische Notation, mit der Petri-Netze gezeichnet werden können (für ein Beispiel siehe Abbildung 2.5). Bei dieser werden Stellen als Kreise dargestellt. Transitionen werden als viereckige Kästen gezeichnet. Die Kanten werden als einfache Pfeile dargestellt. Mit einem Token belegte Stellen werden mittels eines schwarzen Punktes markiert. Kapazitäten werden ebenfalls an die Kreise gezeichnet. Die Kanten erhalten eine Zahl, die angibt, wie viele Tokens durch das Schalten einer Transition aufgebraucht werden und wie viele erzeugt werden. Werden keine Ziffern angegeben, so kann die Anzahl der maximal möglichen Tokens entweder als 1 oder als Unendlich viel angegeben werden (in unserem Beispiel sei dies 1). Sind Kanten nicht beschriftet, so entnehmen die verbundenen Transitionen 1 Token beziehungsweise fügen 1 Token hinzu. In dem in Abbildung 2.5 angegeben Beispiel wäre nur Transition 1 schaltbereit, da nur bei dieser alle eingehenden Stellen (Stelle 1 und Stelle 2) belegt und alle ausgehenden Stellen (Stelle 4) frei sind. Ein Beispiel für eine nicht-bereite Transition wäre Transition 2, da hier die einzige eingehende Stelle (Stelle 3) nicht mit einem Token belegt ist.

2.4 Gears4Net

Das Programmiermodell **Gears4Net** [MS09] wurde für die Entwicklung von massiv parallelen Anwendungen in C# konzipiert. In der Implementierung der Ausführungs-

maschine des neuen Editors dient Gears4Net als Designgrundlage. Im Gegensatz zu Threadprogrammierung, bei der vor allem die Vermeidung von Race Conditions und Synchronisationsprobleme die Entwicklung erschweren, bietet Gears4Net eine Alternative. So werden in Gears4Net keine Threads vom Entwickler programmiert. Gears4Net bietet als Threadalternative sogenannte **Protokolle**. Ein Protokoll versteht sich wie eine eigene kleine Zustandsmaschine mit eigener Nachrichtenschlange, wie die in Abbildung 2.6 dargestellten. Protokolle können mittels dieser Nachrichten empfangen. Es existieren ebenso Möglichkeiten, um Nachrichten an Protokolle zu senden. Mittels vom Entwickler entworfenen, verketteten Wartebedingungen kann ein Protokoll auf das Eintreffen einer oder mehrerer Nachrichten warten. Sind diese Bedingungen erfüllt, kann ein Protokoll in seinen jeweiligen nächsten Zustand gelangen, und dabei dann vom Entwickler erstellten Code ausführen. Um ein Protokoll auszuführen, sind sogenannte **Scheduler** notwendig. Ein Scheduler verwaltet eine beliebige Anzahl von Protokollen und ist primär für die Nachrichtenübergabe und Ausführung der Protokolle zuständig. Im Gegensatz zu Threads brauchen Protokolle und ihre Scheduler keine Kontextwechsel, um aktiv zu werden. Gears4Net setzt, im Gegensatz zur Threadprogrammierung, auf kooperatives Multitasking. So liegt es in der Verantwortung des Protokollprogrammierers, dass Code, den ein Protokoll ausführt, auch irgendwann endet und sich in keiner Endlosschleife verfängt. Zu den großen Vorteilen hierbei zählt, dass die Protokolle, mangels Threadumschaltung, erheblich schneller nacheinander ausgeführt werden können. Durch die Nutzung von mehreren Schedulern, ermöglicht Gears4Net eine deutlich bessere Auslastung von Multi-Prozessor-Maschinen als es die einfache Threadprogrammierung ermöglicht. Während Threads durch Wartebedingungen blockieren, wird bei Gears4Net einfach ein anderes Protokoll ausgeführt. Zu den Nachteilen in Gears4Net zählt unter anderem der, dass ein, sich in einer Endlosschleife befindendes, Protokoll alle seine folgenden, dem selben Scheduler wie es selbst zugeordneten, Protokolle an der Ausführung hindert.

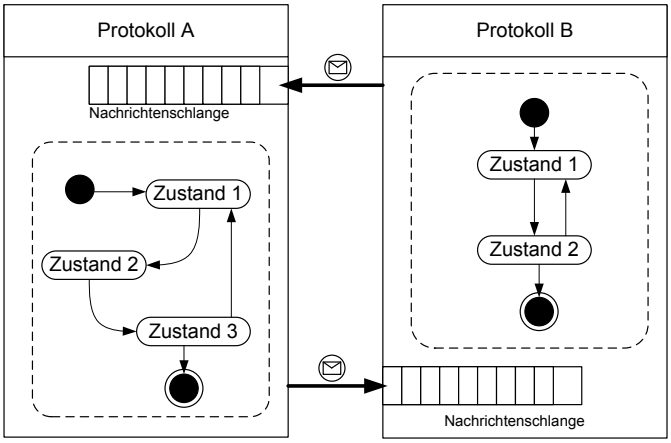

Abbildung 2.6: Beispiel für das Zusammenspiel zwischen Gears4Net Protokollen

2.5 Extensible Markup Language

Die **Extensible Markup Langage** (kurz **XML**) [ERH04] ist eine sogenannte Auszeichnungssprache, mit deren Hilfe hierarchisch Daten strukturiert und dargestellt wer-

den können. In der Implementierung wird XML als Datencontainer für die Persistierung von Pluginketten des neuen Editors genutzt. Grundlage für die Speicherung von XML Dokumenten sind zumeist einfache Textdateien. XML wird häufig genutzt, um Daten plattformübergreifend austauschen zu können. XML Dateien sind durch ihre Struktur sowohl maschinenlesbar als auch menschenlesbar, da sie keine Binärdaten enthalten, außer in Ausnahmefällen. XML Dokumente werden mittels “Elementen” aufgebaut. Ein Element besteht aus sogenannten “Start-Tags” und “End-Tags”. Elemente können zwischen ihren Tags weitere Elemente oder einfach Text enthalten. Ein Beispiel für ein Element wäre <Tag-Name>...< /Tag-Name>. Hier sieht man, dass sowohl Start-Tag als auch End-Tag denselben Tag-Namen besitzen müssen. Ein weiteres wichtiges syntaktisches Element der XML bilden die sogenannten Attribute. Ein Attribut ist ein Schlüsselwort-Wert-Paar, das einem Start-Tag hinzugefügt wird. Ein Beispiel hierfür wäre <Person Geschlecht=“männlich”>...< /Person>. Auf oberster Hierarchieebene erlaubt XML nur ein einziges Element, die Wurzel. Innerhalb der Wurzel lassen sich Elemente aber beliebig verschachteln, wobei die Regeln gelten, dass ein Start-Tag immer durch ein End-Tag geschlossen werden muss und dies ebenentreu geschehen muss. XML-Dokumente besitzen für gewöhnlich eine XML-Deklaration, die allerdings optional ist. Diese Deklaration enthält unter anderem die XML-Version, in der das Dokument verfasst ist. Außerdem kann eine Referenz auf eine sogenannte DTD (Document Type Definiton) angegeben werden, die eine Grammatik enthält, nach der das XML-Dokument aufgebaut ist.

Im Listing 2.1 ist ein einfaches XML-Dokument dargestellt, das eine Liste von Personen beinhaltet. Jede Personen besitzt ein Attribut Geschlecht und jeweils einen Namen und einen Vornamen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Personen>
3          <Person Geschlecht="männlich">
4                  <Name>Müller</Name>
5                  <Vorname>Manfred</Vorname>
6          </Person>
7          <Person Geschlecht="weiblich">
8                  <Name>Meier</Name>
9                  <Vorname>Maria</Vorname>
10         </Person>
11 </Personen>
```

Textauszug 2.1: XML Beispieldokument

Durch ihren einfachen Aufbau lassen sich XML-Dokumente einfach parsen und erstellen. .Net und nahezu jede andere moderne Laufzeitumgebung liefern in ihren Bibliotheken bereits vorgefertigte XMLParser.

3 Konzept und Design

Dieses Kapitel beschreibt zunächst die neue Architektur, die Grundlage für die Entwicklung des neuen Editors ist. Danach beschreibt es die Ideen, die beim Konzept des Models umgesetzt wurden. Diese werden dann mit Hilfe von UML-Modellen so dargestellt, dass sie das Design für die Implementierung des Models aufzeigen. Dann wird ebenfalls konzeptionell beschrieben, wie die Ausführungsmaschine funktioniert, die das Model ausführen kann. Ein UML Aktivitätsdiagramm dient dann als einfaches Design, das darstellt, wie die Ausführungsmaschine arbeitet. Gegen Ende dieses Kapitels wird die Datenübergabe zwischen zwei Plugins beschrieben sowie die Ausführung eines Plugins.

3.1 Architektur des neuen Editors

Wie bereits Eingangs erwähnt, sollte der neue Editor auf einer Model-View-Controller-Architektur basieren. Das Model ist Teil dieser Bachelorarbeit, die View und der Controller sind unabhängig davon in der Bachelorarbeit von Viktor Matkovic beschrieben. Als vierte architektonische Komponente, neben den aus dem MVC-Pattern bekannten Komponenten, wurde die Ausführungsmaschine entwickelt. Der neue Editor wurde, wie in Abbildung 3.1 ersichtlich, in vier Teilen entwickelt. Zum einen der Editor selbst, dann das Model, dann die von Viktor Matkovic entwickelte View mit Controller und die neue Ausführungsmaschine. Das im Folgendem beschriebene Model lässt sich in unterschiedliche einzelne Model-Elemente unterteilen. So muss zum einen der gesamte Workspace modelliert werden. Zum anderen müssen alle in ihm enthaltenen Elemente ebenso modelliert werden. Dazu zählen die Plugins, deren Verbindungen und grafische/textuelle Objekte, die dem Benutzer das Arbeiten mit CT2 einfacher und verständlicher machen sollen. Durch Analyse des bereits vorhandenen, alten Editors, wurden das nun folgende Model für den neuen Editor entwickelt. Unter den Grundlagen, die für die Entwicklung des neuen Editors genutzt wurden, wurde auch eine Funktionsliste in Kapitel 2.1.3 dargestellt, die Funktionen auflistet, die ein neuer Editor bieten muss. Diese Funktionen wurden bei der Entwicklung des neuen Editors ebenso als Grundlage genutzt und innerhalb des Models und mit Hilfe der Ausführungsmaschine umgesetzt.

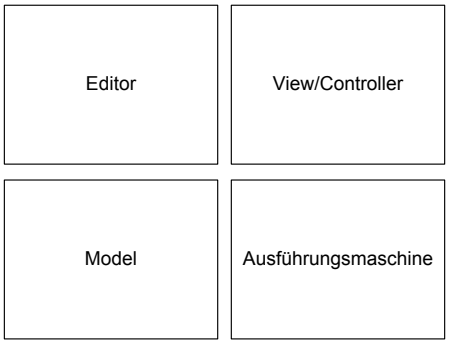

Abbildung 3.1: Architektur des neuen Editors

3.2 Model - Konzept

Das dem neuen Editor zu Grunde liegende Model wurde konzeptionell den bereits beschriebenen Petri-Netzen nachempfunden. Dies birgt erhebliche Vorteile. Petri-Netze sind bereits durch viele Forscher in diversen Arbeiten zur Genüge untersucht worden.

Im Folgenden wird nun beschrieben, wie auf Grundlage eines Petri-Netzes, das Model hinter dem neuen CT2-Editor entwickelt wurde.

3.2.1 WorkspaceModel - das Netz

Das erste Model-Element, das nun beschrieben wird, ist das **WorkspaceModel**. Es stellt den Container dar, in dem alle anderen Model-Elemente verwaltet werden. Im Sinne von CT2 stellt das WorkspaceModel den gesamten Workspace von CT2 dar. Es bietet Möglichkeiten andere Model-Elemente zu erstellen und verwaltet diese dann. Ein neu erstelltes, leeres WorkspaceModel lässt sich anschaulich wie ein leeres Blatt Papier verstehen, auf dem Pluginketten gezeichnet werden können. Im Sinne eines Petrinetzes dient es den Elementen ebenso als Zeichenfläche. Wie die einzelnen Petrinetzelemente auf CT2-Elemente übertragen wurden, beschreiben nun die folgenden Kapitel.

3.2.2 PluginModel - die Transition

Ein **PluginModel** entspricht einer Transition im Petrinetz. Aus CT2-Sicht stellt es ein abstraktes Plugin dar, das auf den Workspace platziert werden kann. Aus diesem Grund beinhaltet jedes PluginModel ein konkretes Plugin. Für Ein- und Ausgaben enthält das PluginModel sogenannte ConnectorModels. Ein ConnectorModel ist genau einem Datenein- oder -ausgang seines enthaltenen Plugins zugeordnet. Ein PluginModel beziehungsweise das in ihm befindliche Plugin kann ausgeführt werden, sobald alle angeschlossenen eingehenden ConnectorModels Daten enthalten und alle ausgehenden, verbundenen ConnectorModels keine Daten enthalten. Das PluginModel ist in diesem Fall schaltbereit.

3.2.3 ConnectorModel - die Stelle

Ein **ConnectorModel** entspricht einer Stelle im Petrinetz. Es kann genau ein Token aufnehmen, womit seine Kapazität erschöpft ist. Durch Schaltung eines PluginModels, werden die Daten aller eingehenden ConnectorModels verbraucht. Sie werden wieder Token-frei gemacht und alle ausgehenden ConnectorModels mit Daten befüllt bzw. mit Tokens belegt. Dadurch, dass ein ConnectorModel nur ein Datum aufnehmen kann und ein PluginModel nur dann ausgeführt werden kann, wenn die Stellen entsprechend belegt sind, ergibt sich durch diesen Ansatz automatisch eine Flusskontrolle innerhalb der Ausführung des Models. Die Ausführung eines Pluginmodels bedeutet, dass das in ihm enthaltene Plugin gestartet wird.

3.2.4 ConnectionModel - die Kante

Da nun Stellen und Transitionen umgesetzt wurden, fehlen im Model nur noch die Kanten, welche sowohl die Stellen (ConnectorModels), als auch die Transitionen (PluginModels) miteinander verbinden. Wo jedoch in einem klassischen Petrinetz die Verbindungen Stelle-Kante-Transition beziehungsweise Transition-Kante-Stelle existieren, verbindet das **ConnectionModel** zwei ConnectorModels miteinander. Ein PluginModel kann sowohl eingehende ConnectorModels, als auch ausgehende ConnectorModels beinhaltet. Somit kann immer ein ausgehendes ConnectorModel eines PluginModels mit einem eingehenden ConnectorModel eines anderen PluginModels verbunden werden. Ein eingehendes ConnectorModel darf jedoch auch mehrere ConnectionModels besitzen, die auf dieses zeigen. Ein abstraktes Beispiel für das beschriebene Model ist in Abbildung 3.2 dargestellt.

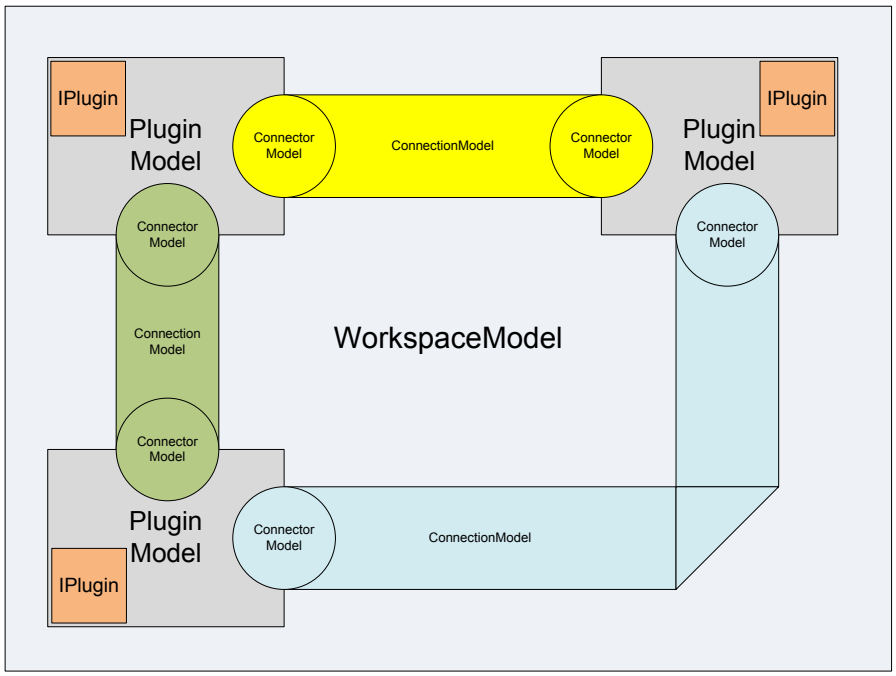

Abbildung 3.2: Beispielzeichnung des Konzepts des Models

3.2.5 TextModel und ImageModel - Texte und Bilder

Da ein WorkspaceModel ohne beschreibende Bilder und Texte unerfahrenen neuen Benutzern ziemlich unverständlich ist, ermöglichen das **TextModel** und das **ImageModel** die Platzierung von Texten und Bildern innerhalb des WorkspaceModels. Texte und Bilder stellen selbst aber nur starre Objekte dar, die keinen Einfluss auf die, im späteren Verlauf beschriebene, Ausführung des Models an sich haben.

3.3 Persistenz - Konzept

Um das Model zu persistieren, sind genau zwei Schritte nötig. Zum einen müssen alle Einstellungen eines Plugins gespeichert und später auch wieder geladen werden. Aus diesem Grund gibt es das **PersistantModel**, welches einerseits ein vollständiges WorkspaceModel enthält, andererseits aber auch alle Einstellungen in sogenannten **SettingModels** speichert. Dieses gesamte Datenpaket wird in ein XML-Format umgewandelt und auf der Festplatte gespeichert, so dass gespeicherte Models menschlich lesbar sind. Das gesamte Model soll durch einen Parser geparst und dann jedes Model-Element in eine XML-Repräsentation umgewandelt werden.

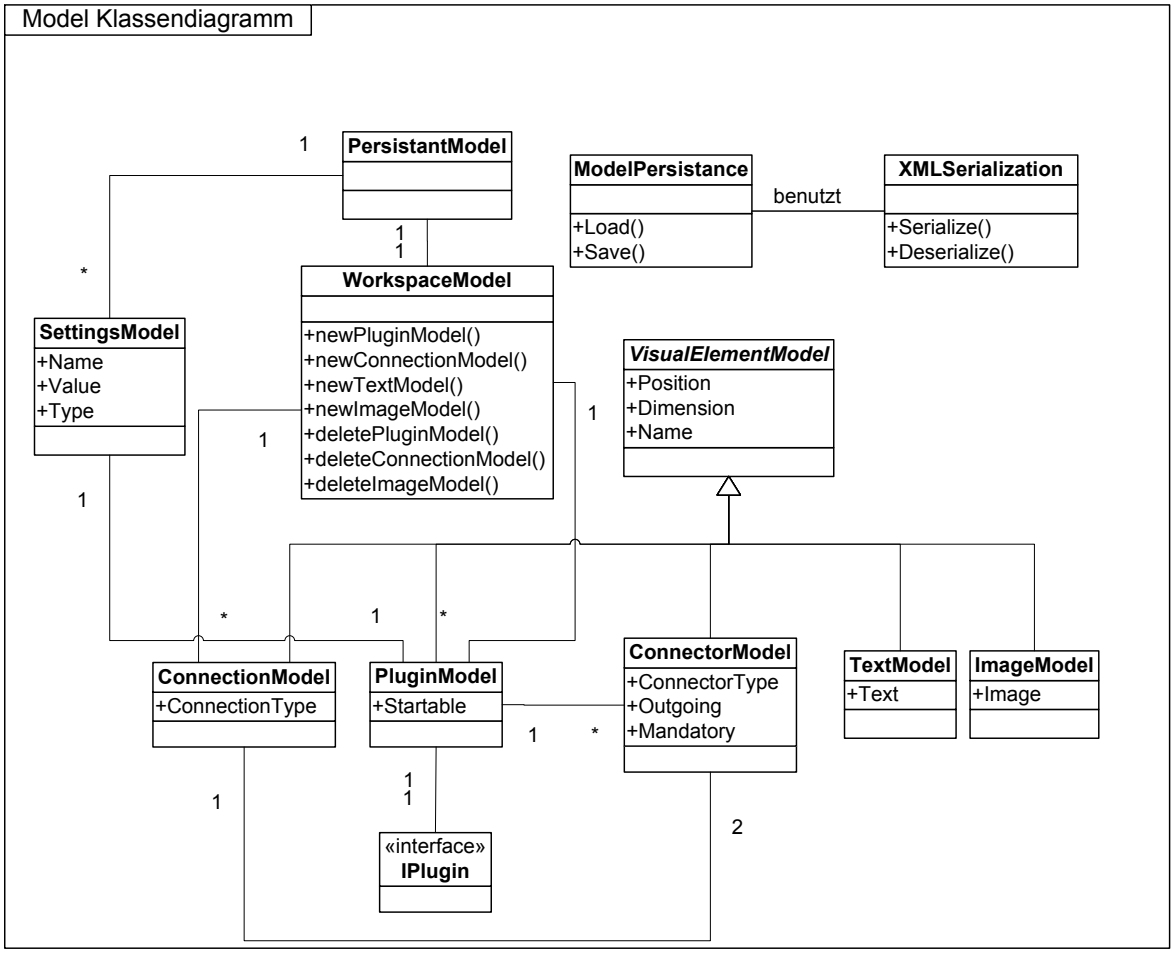

Abbildung 3.3: Klassendiagramm des neuen Models

3.4 Model - Design

Wenn man alle im Konzept beschriebenen Model-Elemente genauer betrachtet und diese in Relationen zueinander setzt, so erhält man das in Abbildung 3.3 dargestellte Klassendiagramm [GB07]. Auf oberster Hierarchiestufe steht die PersistantModel Klasse,

welche einerseits eine Referenz auf ein WorkspaceModel enthält, andererseits eine Liste von SettingsModels, welche Einstellungen von Plugins speichern (Das PersistantModel wird durch die ModelPersistance Klasse beim Speichern erstellt und nur beim Laden benötigt). Die WorkspaceModel Klasse verwaltet ihrerseits Listen auf alle PluginModels, ConnectionModels, TextModels sowie ImageModels. Deren Klassen, sowie die Klasse ConnectorModel, sind abgeleitet von der abstrakten Klasse VisualElementModel, welche Position, Dimension und Namen des jeweiligen Model-Elements enthält. Die PluginModel Klasse verwaltet eine Liste von ConnectorModels und beinhaltet ein Plugin, das von IPlugin abgeleitet ist. ConnectorModel Objekte und ConnectionModel Objekte besitzen einen Datentyp, der den Typ der Daten angibt, welchen sie beinhalten können. Ein ConnectionModel Objekt verbindet immer genau zwei ConnectorModel Objekte (eines mit Outgoing und eines ohne Outgoing Flag gesetzt). Die Erstellung aller Model-Elemente ist durch Methoden des WorkspaceModels (new..., delete...) möglich. Durch die Nutzung dieser Methoden werden alle Verbindungen zwischen den einzelnen Model-Elementen automatisch korrekt durchgeführt. Die ModelPersistance Klasse dient der Speicherung und Ladung eines Models (Load, Save - Methoden). Sie benutzt eine XMLSerialization Klasse, um den C# Objektgraphen eines Models in ein XML und zurück umwandeln zu können (Serialize, Deserialize - Methoden). Die konkrete Implementierung der in diesem ersten Klassenmodell dargestellten Klassen erfolgt in einem späteren Kapitel.

3.5 Ausführungsmaschine - Konzept

Die Ausführungsmaschine ist der Teil des neuen Editors, der das Model interpretiert beziehungsweise, wie der Name schon sagt, “ausführt”. Die Ausführungsmaschine bekommt das Model als Eingabe übergeben und analysiert dieses während der Ausführung. Die Verknüpfung der einzelnen Model-Elemente im Sinne eines Petri-Netzes ermöglicht das Design einer einfachen, intuitiv zu verstehenden Maschine.

Sobald die Ausführungsmaschine gestartet wird, durchläuft diese zunächst den Graphen des Models und sucht PluginModels, die als Startbar markiert sind. Alle PluginModels, die so gefunden werden, dürfen ohne weitere Prüfungen ausgeführt werden. Ausführen eines PluginModels heißt, dass das ihm zugeordnete Plugin ausgeführt wird. Das Plugin wiederum füllt nun alle seine Ausgänge mit Daten. Diese Daten werden von der Ausführungsmaschine wiederum über die, den Ausgängen zugeordneten, ConnectorModels über ConnectionModels an die eingehenden ConnectorModels weiterer PluginModels propagiert.

Nach erfolgreicher Ausführung eines PluginModels werden alle, mit diesem PluginModel verbundenen, weiteren PluginModels ausgeführt, sofern die folgenden Bedingungen erfüllt sind:

- Alle Eingänge des PluginModels, die nicht optional sind, sind mit einem ConnectionModel verbunden
- Alle angeschlossenen Eingänge (ConnectorModels) des auszuführenden PluginModels sind mit Daten befüllt

- Alle, an das auszuführende PluginModel, angeschlossenen Eingänge (ConnectorModels) anderer PluginModels sind frei (d.h. sie sind mit keinen Daten belegt)

Die Ausführung eines Models lässt sich wie Schaltungen eines Petri-Netzes verstehen, wobei die Daten der ConnectorModels als Tokens angesehen werden können. Der Vorteil an diesem Konzept ist der, dass die Ausführungen der PluginModels durch unterschiedliche Threads ausgeführt werden können, da sie, bis auf die drei genannten Bedingungen, für sich alleine gesehen, völlig autark arbeiten können. Die Synchronisation der einzelnen Threads kann mittels Nachrichtenaustausch gewährleistet werden. Sofern ein PluginModel ausführbar ist, erhält es nach der Ausführbarkeitsüberprüfung eine Nachricht und kann ausgeführt werden.

Die Ausführungsmaschine kann zu jeder Zeit angehalten werden, indem in CT2 der Stop-Button geklickt wird. Gestartet wird sie über den Start-Button. Sofern die Maschine gestartet wird, werden alle ConnectorModels auf einen leeren Initialstatus zurückgestellt.

Ein großes Problem im alten Editor und auch in vielen, in CT2 enthaltenen Plugins ist die Darstellung der Ausgaben der Plugins. Hierzu werden WPF-Komponenten benötigt, die immer durch den GUI-Thread von CT2 angesprochen werden müssen. Dafür muss aktiv in den GUI-Thread umgeschaltet werden und das kostet Rechenzeit. Die neue Ausführungsmaschine schaltet automatisch in einem konfigurierbaren Intervall in den GUI-Thread. In diesem durchläuft sie alle Model-Elemente und überprüft, ob deren View sich aktualisieren muss. Sofern dies notwendig ist, wird auf dem Model-Element eine Aktualisierungsmethode aufgerufen, die im GUI-Thread ausgeführt wird. Innerhalb dieser Methode kann das Model seine View aktualisieren beziehungsweise seine View anleiten, sich selbst zu aktualisieren. Der Vorteil bei diesem Vorgehen ist, dass nun nur genau einmal in den GUI-Thread umgeschaltet werden muss und Rechenzeit gespart wird. Der Nachteil ist folgender: GUI-Aktualisierungen einzelner View-Elemente werden nicht mehr zu der Zeit stattfinden, in der sich das Model geändert hat. Durch eine Verringerung der Intervallzeit dieser Aktualisierungen, lässt sich dieses Problem bei Bedarf allerdings minimieren.

3.6 Ausführungsmaschine - Design

Auf Grundlage des entwickelten Konzeptes kann man die Ausführung der Ausführungsmaschine (ohne Betrachtung der GUI) mit dem in Abbildung 3.4 dargestellten Aktivitätsdiagramm [GB07] beschreiben.

Zunächst werden alle PluginModels iteriert und eine, von CT2 festlegte, “PreExcution”-Methode derer Plugins ausgeführt. In dieser Methode kann der Pluginprogrammierer z. B. sein Plugin in einen für die Ausführung notwendigen Zustand versetzen (z. B. bestimmte Ressourcen laden, Handler setzen etc). Sind alle PluginModels durchwandern werden diese erneut durchwandert, um PluginModels zu starten (Aufruf einer “Execute”-Methode des Plugins), die erneut gestartet werden dürfen. Sind alle PluginModels wiederum durchlaufen, wird in einer weiteren Schleife geprüft, ob ein PluginModel erneut gestartet werden darf (siehe Ausführungsbedingungen für PluginModels im Konzept). Des weiteren wird kontinuierlich abgeprüft, ob der Benutzer von CT2 eine weitere Ausführung wünscht. Möchte der Benutzer die Ausführungen beenden,

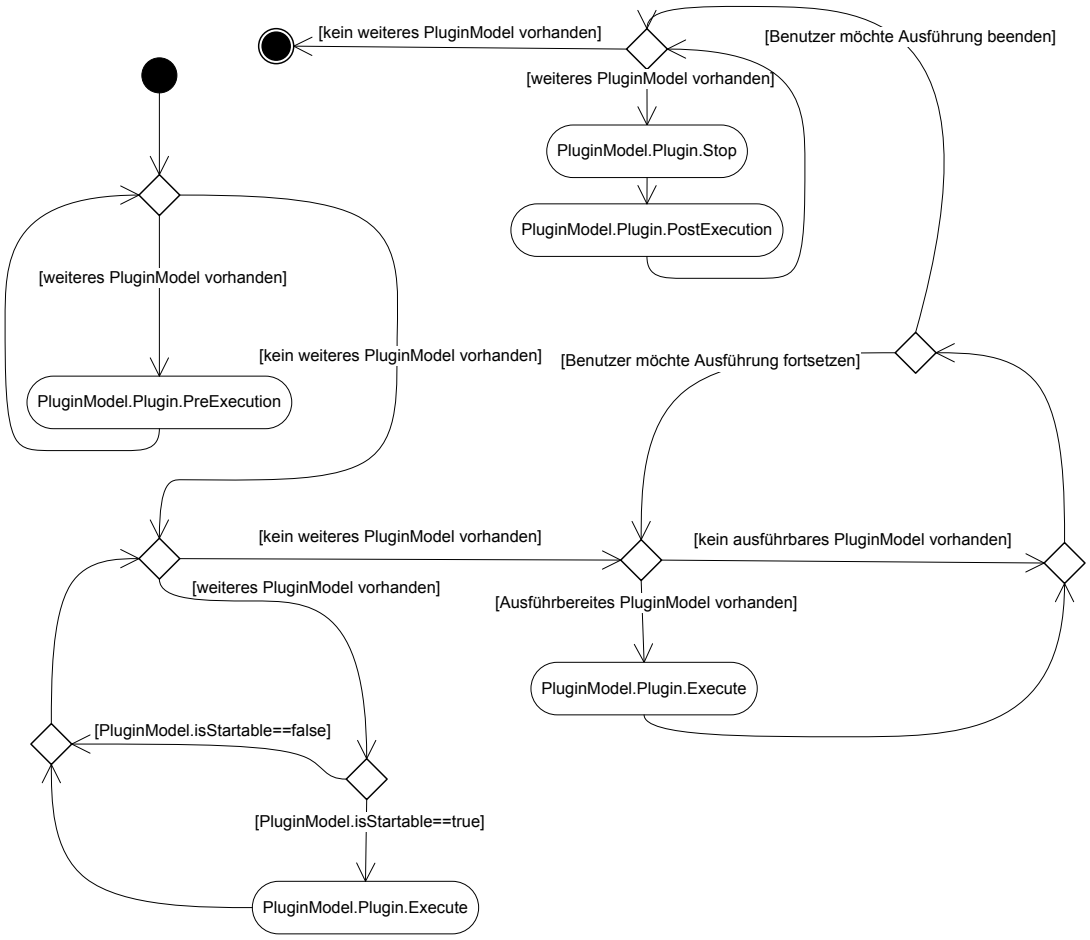

Abbildung 3.4: Aktivitätsdiagramm der Ausführungsmaschine

wird bei jedem Plugin aller PluginModels sowohl eine “Stop”-Methode, als auch eine “PostExecution”-Methode aufgerufen. “Stop”, um dem Plugin mitzuteilen, dass es beendet wird und “PostExecution”, um dem Pluginprogrammierer zu ermöglichen, etwaige Ressourcen wieder frei zu geben.

3.7 Datenübergabe zwischen den Plugins

Wie bereits beschrieben, werden einzelne PluginModels über zwei ConnectorModels, die mittels eines ConnectionModels miteinander verbunden sind, verbunden. Hierfür muss die View Sorge tragen, dass nur ConnectorModels miteinander verbunden sind, die “kompatibel” zueinander sind. Das heißt, dass nur gleiche Datentypen miteinander verbunden werden können (oder Datentypen, die ineinander umgewandelt (“gecastet”) werden können). Dafür bietet das WorkspaceModel eine Hilfsmethode compatibleConnectors(...), die true liefert, sobald zwei übergebene ConnectorModels “kompatibel” sind.

Wurde nun ein Plugin eines PluginModels ausgeführt, so werden die Daten an seine ausgehenden ConnectorModels propagiert. Ein Plugin meldet mittels eines Events, dass es eine Ausgabe gesetzt hat. Das zuständige ConnectorModel reagiert auf dieses Event und speichert die Daten intern ab. Daraufhin belegt es alle seine, über ConnectionModels angeschlossene, ConnectorModels mit diesen Daten.

3.8 Ausführung eines Plugins

Es gibt genau zwei Fälle, in denen ein Plugin eines PluginModels ausgeführt werden kann. Ausführen bedeutet, dass die “Execute”-Methode des Plugins aufgerufen wird. Der erste und einfachere Fall ist der, dass es während des Startvorgangs der Ausführungsmaschine gestartet wird, da es ein Startable-Flag besitzt (siehe Abbildung 3.4). Der zweite Fall ist der, dass alle seine Eingänge mit Daten belegt sind und seine Ausgänge leer sind. Diese Startüberprüfung wird immer genau dann gemacht, wenn ein Eingangs-ConnectorModel mit Daten befüllt wird oder die Daten eines Ausgangs-ConnectorModels “verbraucht” werden. Durch diesen Ansatz erhalten wir eine einfache, auf Petri-Netzen basierende Flusskontrolle, die verhindert, dass ein Plugin zu oft, zu wenig, falsch oder gar nicht ausgeführt wird.

4 Implementierung

Der neue Editor und auch die Ausführungsmaschine sind vollständig mit Hilfe von C# und Microsoft Visual Studio 2010 implementiert worden. Dieses Kapitel beschreibt alle Komponenten, die Teil des neuen Editors sind. Zunächst wird die Umsetzung der Architektur beschrieben und dann auf die einzelnen Komponenten und deren Implementierung in jeweils eigenen Kapiteln eingegangen. Als besonderes Kapitel ist die Persistierung des Models mit Hilfe eines XMLSerializers eingefügt, da seine Entwicklung einen großen Anteil an der Gesamtentwicklung hatte.

4.1 Umsetzung der Architektur des neuen Editors

Der neue Editor wurde in vier Hauptkomponenten aufgeteilt, die jede für sich einzeln implementiert wurde. Abbildung 4.1 zeigt diese Komponenten und ihre Abhängigkeiten in Form eines Komponentendiagramms. Diese Aufteilung war notwendig und birgt einige Vorteile. Zum einen ermöglicht sie die Entwicklung im Team. Während im Umfang dieser Bachelorarbeit das Model und die Ausführungsmaschine entwickelt werden konnte, wurde von Viktor Matkovic parallel dazu die View entwickelt. Durch diese Entwicklung im Team konnte der neue Editor in einer relativ geringen Zeit umgesetzt werden. Ein weiterer Vorteil dieser Komponentenarchitektur mit standardisierten Schnittstellen ist der Vorteil der Austauschbarkeit. Das Model ist so konzipiert, dass es einfach nur über seine Schnittstellen verändert werden kann. Die View nutzt das Model als Blackbox. Theoretisch kann die View angepasst oder sogar ausgetauscht werden und das Model bleibt davon unberührt. Einige Ausnahmen kann es hier natürlich geben, da aus Gründen der Geschwindigkeit z. B. einige Stellen des MVC-Konzepts aufgeweicht worden sind und andere Stellen genau für die, von Viktor Matkovic entwickelte, View konzipiert wurden. Andersrum können Änderungen am Model ohne Einfluss auf die View durchgeführt werden. So ist die Persistierung z. B. absolut losgelöst von der View und kann jederzeit durch eine andere ausgetauscht werden. Ebenso kann die Ausführungsmaschine komplett ausgetauscht werden. Hierzu müssen einige kleinere Änderungen am Model vorgenommen werden.

Die erste Hauptkomponente, die alle weiteren Komponenten "zusammenhält", ist der sogenannte "WorkspaceManagerEditor". Dieser ist eine von dem C# -Interface **IEditor** abgeleitete Klasse. Klassen, die von diesem Interface erben, werden automatisch von CT2 beim Startvorgang initialisiert, wenn sie in einer .Net-Assembly verpackt in das CT2-Pluginverzeichnis gelegt wurde. In Abbildung 4.1 ist dieses Interface als angebotene Schnittstelle vom Typ IEditor dargestellt.

Die nächste wichtige Komponente ist die **View**. Sie besteht aus einer Sammlung von C# -Klassen und WPF-Komponenten und wurde, wie bereits erwähnt, von Viktor Matkovic

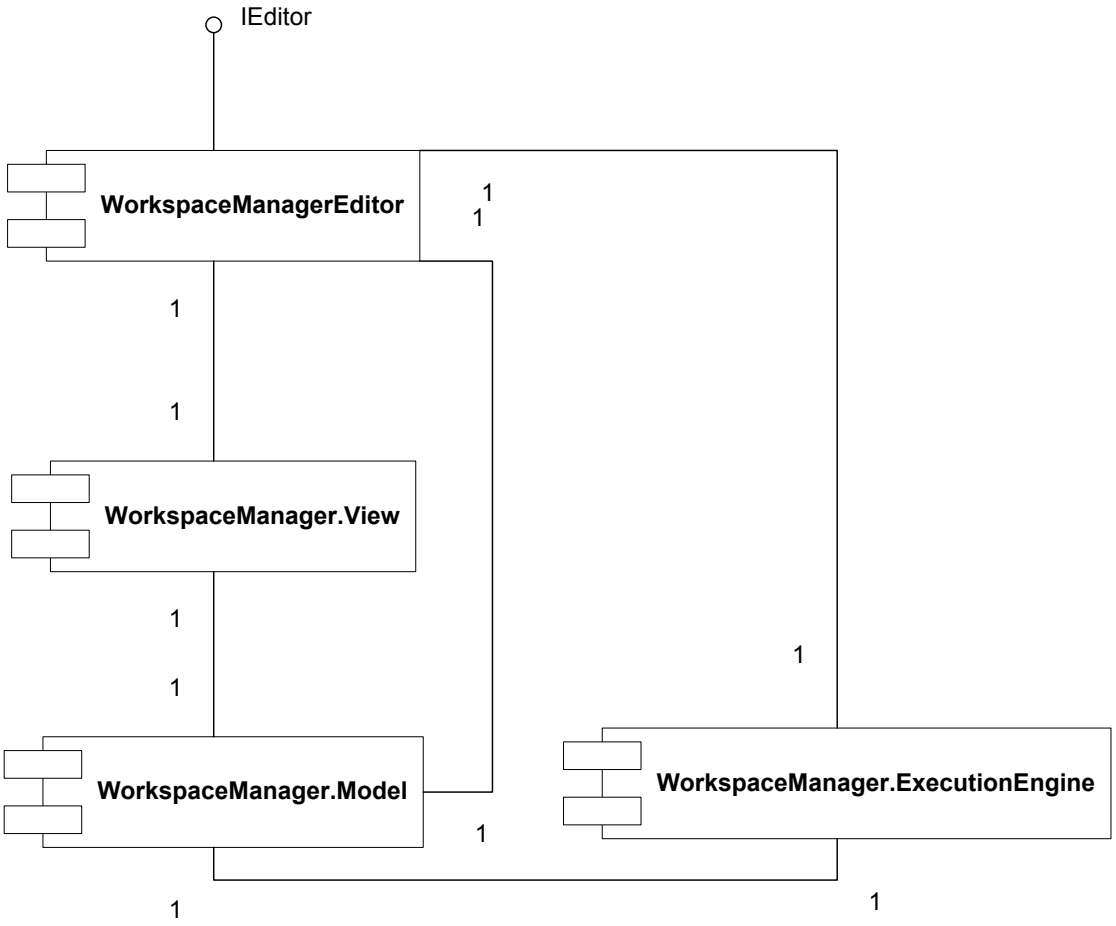

Abbildung 4.1: Komponentendiagramm des neuen Editors

im Rahmen seiner Bachelorarbeit entwickelt. Die View bietet Funktionen, um Pluginketten zu erstellen und zu editieren. CT2 nutzt die View, um den Editor grafisch im Workspace von CT2 darzustellen.

Die dritte Komponente ist das **Model**. Das Model besteht aus einer Menge von C# Klassen, die auf Grundlage des im Konzept aufgezeigten Klassendiagrammes, erstellt wurden.

Die letzte Komponente ist die **Ausführungsmaschine**. Sie wurde mit Hilfe von Gears4Net implementiert und besteht ebenfalls aus einer Reihe von einzelnen Klassen. Die Ausführungsmaschine wurde in ihrem Verhalten nach dem Aktivitätsdiagramm, das im Konzept vorgestellt wurde, umgesetzt.

Im MVC-Muster stellt der Controller eine eigene Komponente dar. WPF bietet dem Entwickler allerdings die Möglichkeit, mit Hilfe sogenannter XAML-Dateien Oberflächen in einer XML-Syntax einfach zu beschreiben. Zu jeder XAML-Datei gibt es eine eigene Partielle Klasse, die C# -Code enthält, und das Verhalten der Oberfläche steuert. Somit kann man den geforderten Controller des MVC-Musters innerhalb der View-Komponente in Form von partiellen Klassen der XAML-Dateien wiederfinden. Der Controller wurde nicht als eigene Komponente implementiert.

Um die einzelnen Komponenten besser zu strukturieren und eine Übersicht zu wahren, welche Subkomponente bzw. Klasse zu welcher Hauptkomponente zugeordnet wird, besitzt jede Hauptkomponente einen eigenen C# -Namespace [1]. Der eigentliche Editor "WorkspaceManagerEditor" befindet sich im Namespace **WorkspaceManager**. Alle Modelklassen befinden sich in **WorkspaceManager.Model**. Die Teile der View wiederum befinden sich in **WorkspaceManager.View** und die der Ausführungsmaschine befinden sich im Namespace **WorkspaceManager.ExecutionEngine**.

4.2 CT2 Schnittstellen

Dieses Kapitel gibt einen Einblick in zwei wichtige C# -Interfaces. Zum einen das **IEditor**-Interface, von dem jeder CT2 Editor ableiten muss und zum anderen das **IPlugin**-Interface, von dem jedes Plugin, das in CT2 geladen werden soll, abgeleitet werden muss.

4.2.1 IEditor

IEditor ist das absolut wichtigste Interface, das als Grundlage für die Neuentwicklung eines Editors für CT2 zwingend genutzt werden muss. Es fordert grundlegende Funktionalitäten, welche ein Editor bereit stellen muss. Außerdem führt die Implementierung des Interfaces erst zur Ladung und Bereitstellung des neuen Editors durch die CT2 Umgebung.

[1] In der Programmiersprache C# , und auch in anderen Sprachen, dienen Namespaces dazu, zusammengehörige Klassen und Objekte innerhalb einer baumartigen Struktur zusammenzufassen. Durch das Importieren eines Namespaces kann der vollständige Ausdruck (Voll-qualifizierter-Name) beim Ansprechen eines Objektes oder einer Klasse weggelassen und der einfache Name genutzt werden

Es fordert die folgenden Properties:

- **CanExecute:** Zeigt an, ob der aktuelle Zustand von CT2 eine Ausführung erlaubt. Ausführung meint hier, die auf seiner Oberfläche dargestellte Kette von Plugins abzuarbeiten
- **CanRedo:** Zeigt an, ob die zuletzt getätigte Aktion des Benutzer, die rückgängig gemacht wurde, wiederholt werden kann
- **CanStop:** Zeigt an, ob die Ausführung der Pluginkette des Editors angehalten werden kann
- **CanUndo:** Zeigt an, ob die zuletzt getätigte Aktion des Benutzers rückgängig gemacht werden kann
- **DisplayLevel:** Ein veraltetes Feature von CT2, das in naher Zukunft entfernt werden soll und unwichtig ist
- **EditorSpecificPlugins:** Liste von Plugins, die speziell nur von diesem Editor genutzt werden können
- **HasChanges:** Zeigt an, ob es Änderungen durch den Benutzer gibt und ob beim Beenden eine Speicherungsaufforderung angezeigt werden sollte

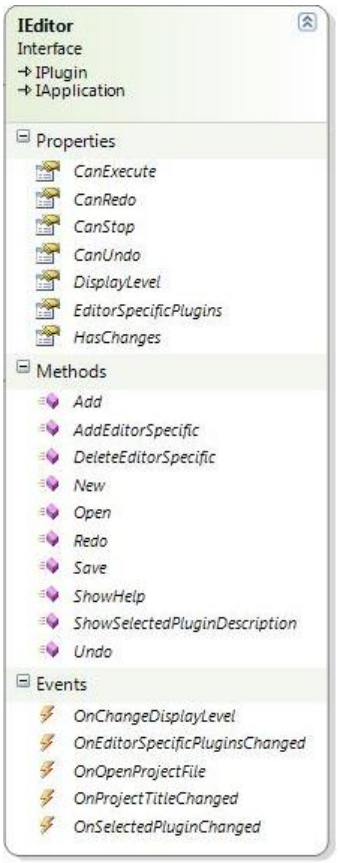

Abbildung 4.2: Das IEditor Interface

Desweiteren fordert das Interface folgende Methoden:

- **Add:** Fügt ein neues Plugin des übergebenen Typs dem Editor hinzu
- **AddEditorSpecific:** Fügt ein neues spezielles, nur dem Editor zugehöriges, Plugin hinzu
- **DeleteEditorSpecific:** Entfernt ein spezielles, nur dem Editor zugehöriges, Plugin
- **New, Open, Save:** Vorhandene Pluginkette des Editors neu erstellen, laden oder speichern
- **Undo, Redo:** Änderung rückgängig machen oder wiederholen
- **ShowHelp:** Zeige die Hilfeseite des Editors
- **ShowSelectedPluginDescription:** Zeig die Beschreibung des ausgewählten Plugins an

Es werden noch folgende Events gefordert:

- **OnChangeDisplayLevel:** Ein veraltetes Event, das in naher Zukunft entfernt werden soll
- **OnOpenProjectFile:** Tritt ein, wenn ein neues Project File geöffnet wurde

- **OnProjectTitleChanged:** Tritt ein, wenn der Name des aktuellen Projekts (Pluginkette) geändert wurde
- **OnSelectedPluginChanged:** Tritt ein, wenn im Editor ein neues Plugin ausgewählt wurde

Das IEditor erbt außerdem von dem im nächsten Unterkapitel vorgestellten Interface IPlugin folgende, wichtige Methoden:

- **Execute:** Startet die Ausführung des vom Editor dargestellten Projekts (Pluginkette)
- **Stop:** Stoppt die Ausführung des vom Editor dargestellten Projekts (Pluginkette)

4.2.2 IPlugin

Ein weiteres sehr wichtiges Interface ist das in Abbildung 4.3 abgebildete. Es bildet die Schnittstelle der bereits mehrfach angesprochenen Plugins. Ein Plugin kapselt ein kryptographisches oder kryptoanalytisches Verfahren oder auch einfache Algorithmen, die z. B. die Texteingabe ermöglichen.

Dazu fordert es folgende Properties:

- **Presentation:** Ein UserControl, welches als "Zeichenfläche" für die graphische Ausgabe des Plugins dient
- **QuickWatchPresentation:** Ein UserControl, welches als "Zeichenfläche" innerhalb des Plugins auf dem Workspace dient
- **Settings:** Eine spezielle Klasse, welche die Einstellmöglichkeiten des Plugins beinhaltet und durch CT2 manipuliert (also das zugehörige Plugin konfiguriert) werden kann

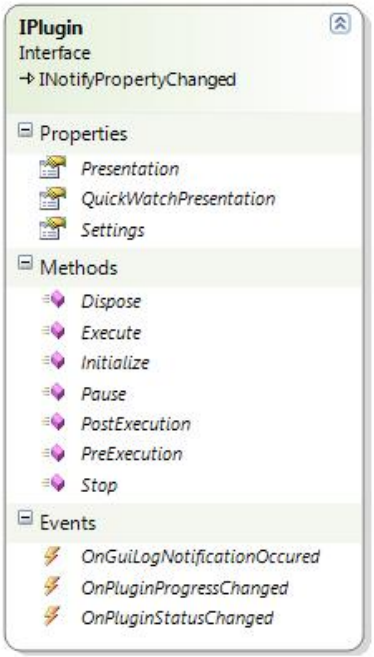

Abbildung 4.3: Das IPlugin Interface

Desweiteren fordert das Interface folgende Methoden:

- **Dispose:** Wird aufgerufen, wenn das Plugin vom Workspace entfernt wird
- **Execute:** Wird aufgerufen, wenn das Plugin ausgeführt werden soll
- **Initialize:** Initialisierungsroutine die aufgerufen wird, wenn das Plugin auf den Workspace gesetzt wird
- **Pause:** Pausiert die Ausführung des Plugins
- **PreExecution:** Wird vor dem Aufruf der Execute Methode aufgerufen
- **PostExecution:** Wird nach dem Aufruf der Execute Methode aufgerufen

- **Stop:** Stoppt die Ausführung des Plugins

4.3 Model

In diesem Kapitel wird die Implementierung des Models beschrieben. Das Model wurde mittels mehrerer Klassen umgesetzt, die nach dem in Abbildung 3.3 gezeigten Design und den im Konzept vorgestellten Ideen, entwickelt wurden.

4.3.1 WorkspaceModel

Die **WorkspaceModel**-Klasse ist die wichtigste Klasse des gesamten Models. Sie verwaltet eine komplette Pluginkette, also im Prinzip den gesamten Workspace, und bietet Hilfsmethoden, um Model-Elemente zu erstellen oder zu löschen. Sie bietet die folgenden öffentlichen Methoden:

```
1  public PluginModel newPluginModel(Point, double, double,
       Type)
2  public PluginModel newPluginModel(Type)
3  public ConnectionModel newConnectionModel(ConnectorModel,
       ConnectorModel, Type)
4  public ImageModel newImageModel(Uri)
5  public TextModel newTextModel()
6  public bool deleteImageModel(ImageModel)
7  public bool deleteTextModel(TextModel)
8  public bool deletePluginModel(PluginModel)
9  public bool deleteConnectorModel(ConnectorModel)
10 public bool deleteConnectionModel(ConnectionModel)
11 public void resetStates()
12 public static bool compatibleConnectors(ConnectorModel,
       ConnectorModel)
```

Textauszug 4.1: Methoden des WorkspaceModels

Mittels der **newPluginModel**-Methode lässt sich ein neues PluginModel erstellen. Eine Überladung der Methode ermöglicht das Erstellen eines PluginModels nur unter Angabe des Plugintyps oder mit Angabe von Positions- und Dimensions-Angaben innerhalb des Workspaces. Durch die Nutzung der Methode wird nicht nur das PluginModel erstellt, es wird außerdem automatisch dem WorkspaceModel zugeordnet und kann über eine öffentliche Liste **AllPluginModels** von außen angesprochen werden. Die View nutzt diese Liste, um für jedes PluginModel eine grafische Repräsemtation zu erstellen.

Die **newConnectionModel**-Methode erstellt ein neues ConnectionModel, das "zwischen" zwei angegeben ConnectorModels steht. Sie erstellt automatisch Verknüpfungen bidirektional zwischen den angegeben ConnectorModels und dem neu erstellt ConnectionModel. Der Aufrufer hat darauf zu achten, dass die ConnectorModels "kompatibel" sind. Das heißt, dass die Datentypen der ConnectorModels die selben oder zumindest ineinander umwandelbar sein müssen. Dazu bietet das WorkspaceModel die später beschriebene Methode *compatibleConnectors*. Das WorkspaceModel verwaltet für alle

ConnectionModels eine öffentliche Liste **AllConnectionModels**, welche auch von der View als Hilfe für die grafische Darstellung genutzt wird.

Die beiden Methoden **newImageModel** und **newTextModel** erstellen jeweils ein neues ImageModel bzw. ein neues TextModel und verlinken diese automatisch mit dem WorkspaceModel. ImageModels und TextModels werden später genauer beschrieben.

Alle **delete***-Methoden ermöglichen das vollständige und saubere Löschen von Model-Elementen innerhalb des WorkspaceModels. Sie lösen alle Verbinden zu etwaigen anderen Model-Elementen und dem zu löschenden Model-Element, um es schließlich zuletzt aus der jeweiligen öffentlichen Liste zu löschen. Ein erfolgreiches Löschen wird mit der Rückgabe des boolschen Wertes “true” signalisiert.

Die **restStates**-Methode ist für die Ausführungsmaschine relevant. Sie ermöglicht das Zurücksetzen aller internen, durch die Ausführungsmaschine gesetzten, Werte der einzelnen Model-Elemente auf einen Standardwert. Darunter fallen z. B. gesetzte Datenwerte von ConnectorModels oder ConnectionModels, die noch nicht durch die Ausführung eines PluginModels “verbraucht” wurden.

Die **compatibleConnectors**-Methode ist eine Hilfsmethode. Sie ermöglicht die Überprüfung, ob zwei übergebene ConnectorModels kompatibel sind. Kompatibel heißt, dass die Datentypen ihrer verwalteten Daten entweder gleich oder ineinander umwandelbar sind. Die Methode liefert den boolschen Wert “true”, falls eine Kompatibilität gegeben ist, andernfalls “false”.

4.3.2 VisualElementModel

VisualElementModel ist die Superklasse für alle zeichenbaren Model-Elemente. Sie besitzt Positions- und Größenangaben. Außerdem bietet sie die Möglichkeit, dem Model-Element einen eigenen Namen zu geben. Die Klasse besitzt ein eigenes Event **onDelete**, das genau dann ausgeführt wird, wenn das VisualElementModel gelöscht wird. Die View nutzt dieses Event, um die Repräsentation beim Löschen seines Model-Elements ebenfalls zu Löschen.

4.3.3 PluginModel

Die **PluginModel**-Klasse repräsentiert genau ein Plugin, das eine Position, eine Größe sowie Connectoren auf dem Workspace bzw. innerhalb des WorkspaceModels besitzt. Diese Positions- und Größenangaben kann es aufnehmen, da es von VisualElementModel abgleitet ist. Ein PluginModel stellt unter anderem die folgenden öffentlichen Methoden, Properties und Attribute zur Verfügung:

```
1 public IPlugin Plugin
2 public Type PluginType
3 public bool RepeatStart
4 public bool Startable
5 public bool Minimized
6 public double PercentageFinished
7 public WorkspaceModel WorkspaceModel
```

```
8  public UserControl PluginPresentation
9  public PluginViewState ViewState
10 public PluginProtocol PluginProtocol
11 public void generateConnectors()
12 public Image getImage()
```

Textauszug 4.2: Methoden, Properties und Attribute des PluginModels

Das **Plugin**-Property zeigt auf genau ein, von IPlugin abgeleitetes Plugin. Dieses Plugin ist das eigentliche Plugin, das somit von dem PluginModel gewrapped wird. Das Plugin kann später durch die Ausführungsmaschine über dieses Property direkt angesprochen werden. Außerdem bietet das PluginModel, mit Hilfe des Properties **PluginType**, direkten Zugriff auf den Typ des Plugins, den es verwaltet. Sobald zum ersten mal auf das Plugin Property zugegriffen wird, wird mit Hilfe von Reflection und dem PluginType eine Instanz des Plugins erstellt. Sobald später wieder auf das Plugin zugegriffen wird, wird wieder dieselbe Instanz zurückgegeben, die beim ersten Zugriff erstellt wurde.

RepeatStart signalisiert der Ausführungsmaschine, ob ein Plugin "von sich aus" ohne neue Dateneingänge gestartet werden darf. **Startable** zeigt generell, ob ein Plugin beim Starten der Ausführungsmaschine als erstes gestartet werden darf. Starten des Plugins heißt, dass eine sogenannte Execute-Methode des Plugins aufgerufen wird. Das dritte Attribut **Minimized** ist nur für die View relevant. Es gibt an, ob ein Plugin in einem sogenannten ikonisierten Zustand gezeichnet werden soll oder nicht. Ikonisiert heißt, dass nicht die Repräsentation des Plugins auf dem Workspace von der View gezeichnet werden soll, sondern nur ein Icon. Das letzte Attribut des PluginModels **PercentageFinished** dient ebenfalls der View. Es sagt aus, wieviel Prozent der Ausführung des internen Plugins bereits abgeschlossen ist.

Das Property **WorkspaceModel** bietet dem Entwickler die Möglichkeit, auf das umgebende WorkspaceModel des PluginModels zugreifen zu können. Die **PluginPresentation** bietet der View die Möglichkeit, auf ein WPF-Usercontrol des Plugins zuzugreifen und dieses als Presentation auf den Workspace zu zeichnen. Der **ViewState** kann sich in einem der drei States (normal, warning, error) befinden. Dies führt zu einer Verfärbung der Repräsentation eines Plugins zu gelb bei "warning" und rot bei "error". Das Property **PluginProtocol** ist für die Ausführungsmaschine relevant. Es ermöglicht dieser, eine Referenz auf ein sogenanntes PluginProtocol, einem Gears4Net Protokoll, zu setzen. Dazu wird in dem Kapitel der Ausführungsmaschine mehr erklärt.

Die Methode **generateConnectors** ist für den Aufbau des Models besonders relevant. Sie analysiert das interne Plugin und generiert sowohl eingehende, als auch ausgehende **ConnectorModels** für jeden Ein- und Ausgang des Plugins. Die generierten ConnectorModels werden in zwei öffentlichen Listen **InputConnectors** und **OutputConnectors** verwaltet. Die Ausführungsmaschine nutzt diese Listen z. B. um Dateneingänge mit Daten zu belegen oder diese auch wieder Daten-frei zu machen.

Die Methode **getImage** liefert das, bereits vorher beschriebene, Icon für den ikonisierten Zustand des Plugins auf dem Workspace.

4.3.4 ConnectorModel

Die **ConnectorModel**-Klasse repräsentiert genau einen Eingang oder einen Ausgang eines PluginModels bzw. des in ihm enthaltenen Plugins. Es stellt unter anderem die folgenden öffentlichen Methoden, Properties und Attribute zur Verfügung:

```
1 public PluginModel PluginModel
2 public WorkspaceModel WorkspaceModel
3 public Type ConnectorType
4 public bool Outgoing
5 public ConnectorOrientation Orientation
6 public bool HasData
7 public Data Data
8 public string PropertyName
9 public void PropertyChangedOnPlugin(Object,
    propertyChangedEventArgs)
```

Textauszug 4.3: Methoden, Properties und Attribute des ConnectorModels

Mittels des **PluginModels** und des *WorkspaceModels* kann der Entwickler auf die, dem ConnectorModel, zugehörigen PluginModel und WorkspaceModel zugreifen.

Der **ConnectorType** liefert den C# -Typ, der Daten, den das ConnectorModel beinhalten kann. Dieser Typ ergibt sich aus dem Typ das Ein- oder Ausgangs, des Plugins des PluginModels, dem das ConnectorModel zugeordnet ist.

Das Attribut **Outgoing** gibt an, ob das ConnectorModel einen Ausgang oder einen Eingang darstellt, indem es “true” oder “false” ist.

Die **Orientation** dient der View als Hilfe. Sie gibt an, ob das ConnectorModel oberhalb, links, rechts oder unterhalb des PluginModels auf den Workspace gezeichnet werden soll.

Das **HasData**-Attribut zeigt an, ob das ConnectorModel derzeit über Daten verfügt. Dies ist für die Ausführungsmaschine relevant, da sie durch dieses Attribut aller ConnectorModels bestimmt, ob ein PluginModel ausgeführt werden darf oder nicht.

Das Attribut **Data** beinhaltet die Daten, die das ConnectorModel seinem PluginModel zur Verfügung stellt.

Der **PropertyName** gibt den Namen des Ein- oder Ausgangs des Plugins des PluginModels an, dem das ConnectorModel zugeordnet ist.

Die Methode **PropertyChangedOnPlugin** wird von dem Plugin des PluginModels aufgerufen, sobald das Plugin ein neues Datum generiert oder ein bestehendes Datum verbraucht hat. Entsprechend wird das Data Attribut gesetzt oder durch “null” ersetzt und das HasData entsprechend gesetzt.

Jedes ConnectorModel besitzt zwei öffentliche Listen **InputConnections** und **OutputConnections** die jeweils eingehende oder ausgehende ConnectionModels beinhalten.

4.3.5 ConnectionModel

Die **ConnectionModel**-Klasse "verbindet" genau zwei ConnectorModels miteinander. Es stellt unter anderem die folgenden öffentlichen Methoden, Properties und Attribute zur Verfügung:

```
1 public WorkspaceModel WorkspaceModel
2 public ConnectorModel From
3 public ConnectorModel To
4 public bool Active
5 public Type ConnectionType
```

Textauszug 4.4: Methoden, Properties und Attribute des ConnectionModels

Das **WorkspaceModel** bietet dem Entwickler Zugriff auf das WorkspaceModel, in dem sich das ConnectionModel befindet.

Die Properties **From** und **To** kennzeichnen den Start- und den Endpunkt des ConnectionModels. Start- und Endpunkte sind immer ConnectorModels.

Das Attribut **Active** signalisiert der View, dass gerade Daten über das ConnectionModel "fließen". Die View kann somit die Repräsentation des ConnectionModel z. B. in einer anderen Farbe zeichnen, um den Datenfluss darzustellen.

Jedes ConnectionModel besitzt einen eigenen Typ **ConnectionType**. Dieser Typ stellt denselben Typ der beiden Enden To und From dar.

4.3.6 TextModel und ImageModel

Die beiden Klassen **TextModel** und **ImageModel** beinhalten Text- und Bildinformationen, die durch die View auf den Workspace gezeichnet werden können. Die Daten werden jeweils in einem internen Byte-Array verwaltet und in ein RichText bzw. in eine JPEG-Grafik umgewandelt, sobald die View diese darstellen möchte. Bilder und Grafiker innerhalb des Workspaces dienen dem Benutzer nur zur Veranschaulichung seiner, von ihm gezeichneten, Pluginketten. Für die Ausführung des Models sind sie unerheblich. Beide Klassen sind, so wie alle Model-Klassen, vom VisualElementModel abgeleitet.

4.4 Persistenz

Dieses Kapitel beschreibt die für die Persistierung aller Model-Elemente notwendigen Schritte. Für diese wurde ein eigener XML-Serialisierer und -Deserialisierer geschrieben. Dieser wandelt die C# -Objekte des Models in eine XML-Struktur um und wieder zurück. Im Laufe der Entwicklung stellte sich heraus, dass XML-serialiserte Model-Elemente jedoch zu Dateigrößen führen, die mehrere Megabytes überschreiten können. Aus diesem Grund wird das XML-Modell vor der finalen Speicherung mit einem GZip-Packer gepackt, was wieder zu Größen in einigen wenigen Kilobytes führt. Der Grund hierfür ist der, dass XML-Dokumente fast ausschließlich aus redundanten Informationen bestehen, die für das Packen prädestiniert sind. Der mitgelieferte .Net XML-Serialisierer

erwies sich als untauglich, da dieser Zirkelbezüge innerhalb der, in ein XML umzuwandelnden Objekte, nicht unterstützt. Außerdem rät Microsoft: "Bei der XML-Serialisierung werden keine Methoden, Indexer, private Felder oder schreibgeschützte Eigenschaften (mit Ausnahme von schreibgeschützten Auflistungen) konvertiert. Verwenden Sie statt der XML-Serialisierung BinaryFormatter, wenn alle öffentlichen und privaten Felder und Eigenschaften eines Objekts serialisiert werden sollen." [msd10]. Da der BinaryFormatter aber nur nicht menschenlesbare Binärdaten produziert und ebenso private Felder serialisiert werden sollen, wurde ein eigener XML-Serialisierer geschrieben.

4.4.1 XMLSerialization

Die Klasse **XMLSerialization** bietet statische Methoden um einen C# -Objektgraphen in ein XML Dokument umzuwandeln und so erstellte XML-Dokumente wieder in C# -Objektgraphen umzuwandeln. Sie besitzt die folgenden Methoden:

```
1 public static void Serialize(object, string, bool)
2 public static void Serialize(object, string, Encoding,bool)
3 public static void Serialize(object, StreamWriter, bool)
4 public static object Deserialize(XmlDocument)
```

Textauszug 4.5: Methoden der Klasse XMLSerialization

Die überladene Methode **Serialize** ermöglicht das Serialisieren eines C# -Objektgraphen. Als Parameter kann der Objektgraph, ein Pfad für die zu erstellende XML-Datei und ein boolscher Wert angegeben werden. Der Objektgraph ist das "Wurzelobjekt", von dem aus alle referenzierten, serialisierbar markierten, Objekte in XML-Syntax umgewandelt werden. Die Serialisierung arbeitet rekursiv auf allen referenzierten Objekten. Der boolsche Wert gibt an, ob die Ausgabedatei ein XML-Dokument im Textformat oder eine, mittels GZIP gepackte, Binärdatei sein soll. Alternativ kann als Ziel auch ein StreamWriter Objekt angegeben werden. Eine weitere Überladung ermöglicht es, die XML-Datei in einer anderen Codierung als die Standardcodierung zu codieren. Als Standardcodierung ist UTF-8 voreingestellt. Im nachfolgenden Kapitel wird beispielhaft eine XML-Datei aufgezeigt, die mittels XML-Serialisierung serialisert wurde.

Die Methode **Deserialize** ermöglicht das Umwandeln einer, mittels Serialize erstellten, XML-Datei in einen C# Objektgraphen. Der so deserialisierte Objektgraph bzw. dessen "Wurzel" wird als Ergebnis der Methode zurückgegeben. Als Eingabe erwartet die Methode ein Objekt des Typs XmlDocument, der innerhalb der C# -Standardbibliothek mitgeliefert wird.

4.4.2 XML-Format

In diesem Kapitel wird beispielhaft gezeigt, wie mittels XMLSerialization.Serialize(...) ein Objektgraph in ein XML-Dokument umgewandelt wird. Der XMLSerializer ist generisch konzipiert und nicht nur für die Serialisierung des Models geeignet. Aus diesem Grund wird das folgende vereinfachte Beispiel verwendet:

```
1  using XMLSerialization;
2
3  class Main{
4
5   static void Main(string[] args)
6   {
7      Personen personen = new Personen();
8      Person p1 = new Person("Hans Hansen");
9      Person p2 = new Person("Peter Petersen");
10     personen.Add(p1);
11     personen.Add(p2);
12
13     XMLSerialization.Serialze(personen,"test.xml",false);
14  }
15
16 }
```

Textauszug 4.6: Einfachs Code Beispiel mit zu serialisierenden Objekten

Der XMLSerializer durchwandert durch den Aufruf in Zeile 13 des Codebeispiels 4.6 das übergebene Personenobjekt. Das Personenobjekt verwaltet eine einfache Liste von Personen, die zwei Einträge ("Hans Hansen" und "Peter Petersen") enthält. Das XML-Wurzelelement eines XML serialisierten Dokuments ist immer ein **objects**-Tag, das angibt, dass das XML-Dokument serialisierte Objekte enthält. Jedes Objekt wiederum wird unter einem eigenen **object**-Tag in die XML-Datei geschrieben. Die Objekte besitzen eine eindeutig generierte ID und ihren C# -Typ. Im Beispiel wird das Personen-Objekt in einen eigenen Eintrag des Typs Personen umgewandelt. Da das Personen-Objekt eine generische Liste des Typs **Person** besitzt, erhält das XML-Personen-Objekt ebenso einen Member mit Namen Liste und des Typs Person. Die Einträge der Liste werden in der XML mittels **reference**-Tag mit weiteren object-Einträgen verknüpft. Da die Liste im Beispiel zwei Person-Objekte beinhaltet, werden unter objects ebenfalls zwei weitere object-Einträge erstellt, die als Member zwei Strings mit den Namen der Personen enthalten:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <objects>
3          <object>
4          <type>Personen</type>
5          <id>1</id>
6          <members>
7                  <member>
8                          <name>Liste</name>
9                          <type>System.Collections.Generic.
                              List;Person</type>
10                         <list>
11                                 <entry>
12                                         <type>Person</type>
13                                         <reference>2</
                                              reference>
```

```
14                                  </entry>
15                                  <entry>
16                                          <type>Person</type>
17                                          <reference>3</
                                               reference>
18                                  </entry>
19                          </list>
20                  </member>
21                  <member>
22                          <name>Anzahl</name>
23                          <type>System.Int32</type>
24                          <value>2</value>
25                  </member>
26          </members>
27          </object>
28          <object>
29          <type>Person</type>
30          <id>2</id>
31          <members>
32                  <member>
33                          <name>Name</name>
34                          <type>System.String</type>
35                          <value>Hans Hansen</value>
36                  </member>
37          </members>
38          </object>
39          <object>
40          <type>Person</type>
41          <id>3</id>
42          <members>
43                  <member>
44                          <name>Name</name>
45                          <type>System.String</type>
46                          <value>Peter Petersen</value>
47                  </member>
48          </members>
49          </object>
50  </objects>
```

Textauszug 4.7: XML Beispieldokument des XMLSerializers

4.4.3 Speichern und Laden im neuen Editor

Für das Speichern und Laden von Pluginketten im neuen Editor sind mehre Schritte notwendig. Beim Speichern werden zunächst alle Einstellungen der einzelnen Plugins geparsed und in eine Liste geschrieben. Diese Liste wird dann über eine sogenannte **PersistantModel** referenziert die auch eine Referenz auf das WorkspaceModel besitzt.

Dieses PersistantModel wird dann mittels des zuvor beschrieben XMLSerializers in eine XML-Datei umgewandelt. Analog funktioniert das Umwandeln eines XML-Dokumentes in eine C#- Objektstruktur genau entgegengesetzt.

Innerhalb CT2 kann der Benutzer die Speichern- und Ladefunktionen in der oben in der Oberfläche angebrachten Werkzeugleiste aufrufen. Sowohl beim Speichern als auch beim Laden können Zieldatei bzw. Quelldatei über einen Dateiauswahldialog gewählt werden.

XML-Dokumente die aus WorkspaceModels bzw. PersistantModels entstanden sind, können, nicht zuletzt durch enthaltene Bilder sehr groß werden (über 1 Megabyte). Da XML allerdings, wie bereits vorher erwähnt, ein sehr stark packbares Format darstellt, werden die XML Dokumente mittels eines GZip-Algorithmus gepackt respektive entpackt.

4.5 Ausführungsmaschine

Die Ausführungsmaschine ist die Komponente des neuen Editors, welche die auf dem Workspace gezeichneten Pluginketten ausführt. Genau genommen wird das Model, das durch die Arbeit des Benutzers, mit Hilfe des Editors, erstellt wurde, ausgeführt. Die Ausführungsmaschine besteht aus sechs Klassen, die im Folgenden nun beschrieben sind.

Die Klasse **ExecutionEngine** hält die Ausführungsmaschine zusammen. Sie bietet die folgenden Methoden:

```
1 public bool IsRunning()
2 public void Execute(WorkspaceModel, int)
3 public void Stop()
4 public void GuiLogMessage(String, NotificationLevel)
```

Textauszug 4.8: Die Schnittstelle der Klasse ExecutionEngine

Mittels den **Execute**- und **Stop**-Methoden kann die Ausführung eines, der Execute-Methode übergebenen Models gestartet bzw. gestoppt werden. Die Zahl der dafür vorgesehen Threads kann mittels der, der Execute-Methode übergebenen, Integer-Variable festgelegt werden.

Die Methode **IsRunning** zeigt dem umgebenden Editor, ob die Ausführungsmaschine sich gerade in der Ausführung befindet oder nicht.

Mittels der öffentlichen Methode **GuiLogMessage** können die weiteren Komponenten der Ausführungsmaschine das Nachrichtensystem von CT2 nutzen und eine Nachricht und ein Dringlichkeitslevel dieser Nachricht übergeben.

Sobald die Ausführungsmaschine gestartet wurde, erstellt Sie einen **WorkspaceManagerScheduler**. Dieser Scheduler ermöglicht der Ausführungsmaschine, Gears4Net Protokollinstanzen auszuführen. Im Gegensatz zu den, im Gears4Net bereits mitgelieferten Schedulern, bietet der WorkspaceManagerScheduler einige Neuerungen. Zum einen nutzt der Scheduler Multitasking. Das heißt, dass die Anzahl der Threads, die der Scheduler für

das Scheduling der Gears4Net Protokolle nutzt, vom Benutzer des neuen Editors festgelegt werden kann. Des weiteren bietet der WorkspaceManagerScheduler die Möglichkeit, die Priorität seiner Threads festzulegen. Der Scheduler durchläuft in einer Schleife eine Warteschlange, in der sich ausführbare Gears4Net Protokolle befinden. Diese werden der Reihe nach aus der Warteschlange entnommen und ausgeführt. Diese Arbeit kann, wie bereits erwähnt, von mehreren Threads gleichzeitig ausgeführt werden.

Nachdem der WorkspaceManagerScheduler gestartet wurde, wird von der Ausführungsmaschine für jedes PluginModel ein sogenanntes **PluginProtocol** erstellt.

Das PluginProtocol ist ein Gears4Net Protokoll, das auf sogenannte **ExecutionMessages** reagiert. Sobald ein PluginProtocol eine ExecutionMessage zugestellt bekommt, überprüft es zunächst, ob das ihm zugeordnete PluginModel, ausgeführt werden darf. Hierzu durchwandert es alle notwendigen ConnectorModels und überprüft, ob die eingehende ConnectorModels Daten besitzen und die ausgehenden ConnectorModels Datenfrei sind. Ist dies der Fall, so wird das Plugin des PluginModels mit "Daten versorgt". Dies geschieht, indem die Ein- und Ausgänge des Plugins mittels Reflection mit den Daten der ConnectorModels belegt werden. Sobald das Plugin die Daten erhalten hat, wird die Execute-Methode des Plugins ausgeführt. Die Steuerung wird nun dem eigentlichen Plugin übergeben. Nachdem das Plugin ausgeführt wurde, erhalten alle über eingehende ConnectorModels erreichbare, PluginModels eine ExecuteMessage, da sie gegebenenfalls auch ausgeführt werden können.

Neben dem PluginProtocol startet die Ausführungsmaschine ein weiteres wichtiges Gears4Net Protkoll. Das sogenannte **UpdateGUIProtocol** dient dazu, der View Rechenzeit zu geben, sobald GUI-Elemente aktualisiert werden müssen. Dazu besitzt ein jedes Model-Element ein Flag **GuiNeedsUpdate**. Das UpdateGuiProtocol durchwandert in zyklischen, vom Benutzer konfigurierbaren Abständen, das Model und sucht nach Elementen, deren GuiNeedsUpdate-Flag gesetzt wurde. Bei diesen wird dann, auf deren zugeordneten View-Elementen, eine **update**-Methode aufgerufen. Diese Methode veranlasst dann die Aktualisierung der View. Da View-Elemente aber im Kontext des sogenannte GUI-Threads stehen, "marshalled" sich das GuiUpdateProtocol einmalig, am Anfang seiner Ausführung, in diesen. Nur durch dieses Marshalling ist es dem GuiUpdateProtocol möglich über die update-Methoden die View-Elemente zu aktualisieren. Die update-Methoden wurden von Viktor Matkovic innerhalb seiner Bachelorarbeit umgesetzt.

Um die Geschwindigkeit (ausgeführte Plugins pro Sekunde) zu messen, gibt es noch ein weiteres Gears4Net Protokoll. Das **BenchmarkProtocol** bietet die Möglichkeit, sofern es durch den Benutzer eingeschaltet wird, die aktuelle Geschwindigkeit der Ausführungsmaschine anzeigen zu lassen. Das Protocol zählt die erfolgreiche Ausführung von Plugins von PluginProtocols und gibt diese sekündlich aus.

Wie bereits vorher beschrieben, schicken PluginProtocols an alle vorgelagerten PluginProtocls des gerade ausgeführten PluginProtocols sogenannte ExecutionMessages, da diese eventuell wieder ausgeführt werden können. Das könnte z. B. der Fall sein, wenn ihre Ausgänge durch die Ausführung nachfolgender PluginModels wieder frei sind und ihre Eingänge bereits neue Daten enthalten. Es gibt aber noch zwei weitere Möglichkeiten, wie ein PluginModel eine ExecutionMessage erhalten kann. Zum einen sendet die Ausführungsmaschine an jedes PluginProtocol, dessen PluginModel ein Startable-Flag

besitzt, während des Startvorgangs der Ausführungsmaschine eine initiale **Execution-Message**. Dies bewirkt, dass initial alle startbaren Plugins am Anfang gestartet werden. Die weitere Möglichkeit, durch die eine ExecutionMessage versendet werden kann, ist ein ConnectorModel. Sobald ein ConnectorModel über, den ihm zugeordneten Datenausgang eines Plugins, Daten erhält, versorgt es alle weiteren PluginModels, die über Connection- und ConnectorModels mit ihm verbunden sind, eben mit diesen Daten. Nachdem es alle seine Nachfolger mit Daten versorgt hat, verschickt es an diese ExecutionMessages. Das führt dazu, dass Nachfolger, die mit ausreichend Daten versorgt und deren Ausgänge "frei" sind, ebenfalls durch die Ausführungsmaschine ausgeführt werden.

5 Evaluation

Die Evaluation soll zeigen, ob und wie der neue Editor die an ihn gestellten Anforderungen umsetzt. Zunächst wird die Kompatibilität zum alten Editor gezeigt. Hierzu wird gezeigt, dass der neue Editor sowohl alle bereits bestehenden Plugins genau so wie der alte Editor ausführt. Des Weiteren wird in Kapitel Messungen sowohl die Geschwindigkeit des neuen Editors, in Form von ausgeführten Plugins in der Sekunde, als auch die Speichereffizienz dargestellt. In der Auswertung wird dann dargestellt, inwiefern die einzelnen Anforderungen umgesetzt worden sind oder nicht.

5.1 Kompatibilität zum alten Editor

CT2 bietet bereits eine große Anzahl an fertig entwickelten Plugins. Diese sind größtenteils bereits getestet und durch die CT2-Entwickler für den praktischen Gebrauch freigegeben. Die in der Tabelle 5.1 aufgeführten Plugins wurden mit dem neuen Editor bereits getestet. Für sie wurde bereits ein sogenanntes Sample erstellt. Ein Sample ist eine abgespeicherte Pluginkette, die dem Benutzer beispielhaft zeigen soll, wie einzelne Plugins in CT2 zu gebrauchen sind. Diese Samples befinden sich in einem, mit CT2 mitgelieferten, Verzeichnis und können durch den Benutzer in CT2 geladen werden. CT2 umfasst mittlerweile ca 100 Plugins, die alle einmal in den neuen Editor geladen, aber nicht komplett getestet worden sind. Die in Tabelle 5.1 aufgeführten Plugins bilden eine Auswahl von getesteten Plugins. Sie beinhaltet einige kryptographische, einige weniger komplexe und einige kryptoanalytische Plugins, die ohne Probleme im neuen Editor funktioniert haben.

5.2 Messungen

Das folgende Kapitel stellt einige Messungen vor, die im neuen Editor vorgenommen wurden. Diese Messungen sollen die mögliche Ausführungsgeschwindigkeit des neuen Editors belegen. Außerdem sind einige Messungen zur Speichereffizienz des neuen Editors durchgeführt worden. Im Zuge der Geschwindigkeitsmessung soll auch belegt werden, dass der neue Editor bzw. die neue Ausführungsmaschine Pluginketten deutlich schneller als der alte Editor ausführt.

5.2.1 Wie gemessen wurde

Um die Geschwindigkeit und die Speichereffizienz der Ausführungsmaschine zu ermitteln wurde ein spezielles Gears4Net Protokoll in selbige eingebaut. Das bereits vorher einmal angesprochene **BenchmarkProtocol** (siehe Kapitel 4) misst die aktuelle Geschwindigkeit der Ausführungsmaschine und gibt diese in “ausgeführten Plugins pro Sekunde”

an. Der aktuell von CT2 reservierte Arbeitsspeicher wird ebenso gemessen. Diese beiden Werte werden mittels zweier Textdateien ausgegeben und können dann einfach, z. B. mit einer Tabellenkalkulation weiter verarbeitet werden. An dieser Stelle fiel auf, dass die Messungen das Ergebnis etwas verfälschen, da Messungen sowohl Geschwindigkeit kosten als auch Speicher benötigen. Um jedoch Vergleiche zwischen verschiedenen Systemen und Konfigurationen zu ziehen, waren die Messungen jedoch ausreichend, da diese Verfälschung der Messergebnisse zwischen 5 und 10 Prozent liegen. Dies ließ sich durch Beobachtung des Taskmanagers während der Ausführung mit und ohne eingeschaltetes BenchmarkProtocol erkennen und abschätzen.

5.2.2 Ausführungsgeschwindigkeit

Mittels der Geschwindigkeitsmessung soll zum einen gezeigt werden, dass der neue Editor bzw. seine Ausführungsmaschine schnell sind. Zum anderen soll gezeigt werden, dass das Einschalten von mehreren Prozessoren bzw. Kernen auch einen Geschwindigkeitsgewinn hervorbringt. Die Hinzuschaltung von mehreren Kernen wird vom neuen Editor explizit unterstützt. Der Geschwindigkeitsgewinn wurde im Folgenden ermittelt. Ein direkter Messvergleich der Geschwindigkeit des neuen mit dem alten Editor ist nicht ohne Weiteres möglich, da der alte Editor über keinerlei Messvorrichtungen verfügt. Jedoch ist die Ausführung im alten Editor gerade noch so schnell, das man die Datenübergaben zwischen einzelnen Plugins mit dem menschlichen Auge wahrnehmen kann. Dadurch lässt sich grob abschätzen, dass der alte Editor eine Ausführungsgeschwindigkeit von einigen zehn bis hundert Plugins pro Sekunde verfügt.

Um die Ausführungsgeschwindkeit zu messen, wurde eine Pluginkette mit mehreren einfachen Plugins erstellt, die jedes für sich innerhalb von Millisekunden ausgeführt werden können. Die Pluginkette besitzt mehrere Schleifen, um eine kontinuierliche Ausführung zu ermöglichen. Diese Pluginkette wurde gespeichert und für jede Messung als Grundlage gewählt. Durch die Wahl von sehr einfachen Plugins minimiert sich die, von den Plugins benötigte Rechenzeit, auf ein Minimum. Die gemessenen Werte liegen dadurch relativ nah an denen der reinen Ausführungsmaschine. Damit auch die volle Auslastung der Ausführungsmaschine erreicht wird, wurden mehrere Schleifen innerhalb der Kette konstruiert. Einfachere Tests mit weniger Plugins haben gezeigt, dass die Ausführungsmaschine mit wenigen Plugins nicht voll ausgelastet werden kann. Dies kommt daher, da nicht genügend ausführbereite Plugins bereit stehen, die der Scheduler ausführen könnte.

Im Folgenden wird zunächst die Ausführungsgeschwindigkeit mit nur einem Prozessor auf einem handelsüblichen Desktop PC betrachtet. Wie aus dem Graphen in Abbildung 5.1 ersichtlich, erreicht der neue Editor eine mittlere Ausführungsgeschwindigkeit von ca. 81.500 Plugins/s, dargestellt durch die rote Linie. Die schwarze Linie zeigt den Geschwindigkeitsverlauf gemessen über eine Stunde. Wie zu erkennen ist, benötigt die Ausführungsmaschine relativ kurze Zeit, um die volle Ausführungsgeschwindkeit zu erreichen. Über die Zeit gesehen zeigt sich ein relativer konstanter Geschwindigkeitsverlauf, der nur durch vereinzelte Ausreißer unterbrochen wird. Diese Ausreißer lassen sich z. B. durch den Anlauf des C#-GarbageCollectors erklären. Dieser ist ein besonderer Thread, der in jeder .Net-Anwendung im Hintergrund tätig ist und den Speicher aufräumt (nicht

referenzierte Objekte löscht). Diese “GarbageCollection” benötigt einige Zeit und bremst somit die Ausführung kurzzeitig.

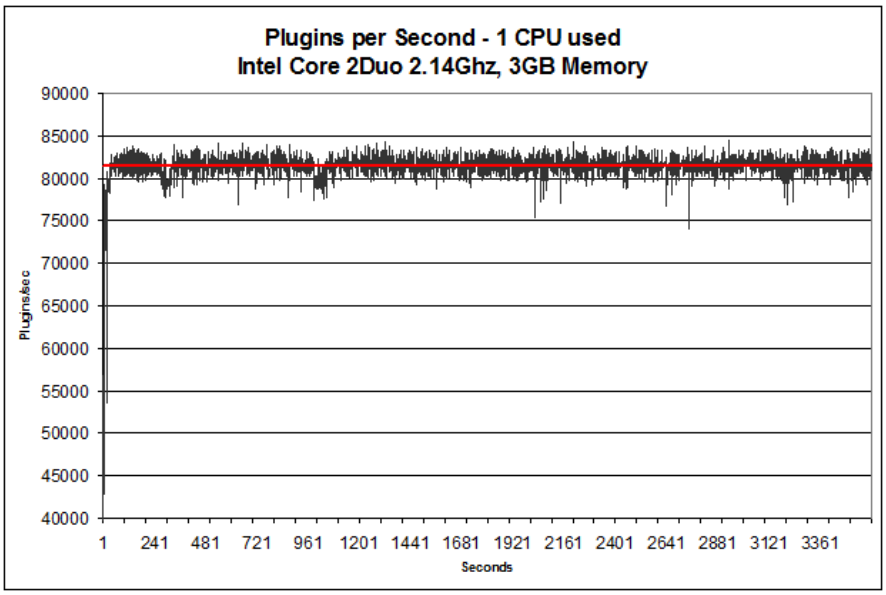

Abbildung 5.1: Geschwindigkeit der Ausführungsmaschine mit einem Prozessor

In Abbildung 5.2 ist ähnlich wie in Abbildung 5.1 die Geschwindigkeit der Ausführungsmaschine bei Nutzung von zwei Prozessoren dargestellt. In dieser Messung wurde eine durchschnittliche Ausführungsgeschwindkeit von 122.300 Plugins/s ermittelt. Die deutlichen Ausreißer in dem Graphen lassen sich wieder durch den GarbageCollector erklären. Andererseits lassen diese sich auch durch Messfehler erklären. Da die .Net-Plattform und Windows keine Echtzeit bieten und unter Volllast manche Zeitgeber verfrüht oder verspätet agieren, kann es sein, dass zwei Messungen direkt hintereinander stattfinden. Das führt dazu, dass einige Messungen einen unrealistischen Wert nahe der Null liefern. Genauso ist es möglich, dass Hintergrunddienste von Windows die Ausführung kurzzeitig zum blockieren bringen und somit in dem gemessenen Intervall wirklich keine oder nur wenige Plugins ausgeführt wurden.

Durch Vergleich verschiedener Messungen und mehrfacher Wiederholungen, auch über die in den beiden Graphiken dargestellten Zeitverläufen hinaus, konnte ermittelt werden, dass der Sprung von einem auf zwei Prozessoren in der aktuellen Ausführungsmaschine einen Geschwindigkeitsgewinn von ca. 50% ermöglicht.

5.2.3 Speichernutzung

Da kryptoanalytische Verfahren häufig mehrere Stunden, Tage oder Monate stabil durchlaufen werden müssen, um ein Ergebnis zu liefern, wurde eine Analyse des Speicherverbrauchs von CT2 während der Ausführung der Ausführungsmaschine des neuen Editors durchgeführt. Diese soll sicherstellen, dass keine Speicherlücken vorhanden sind und CT2 und der neue Editor stabil über lange Zeit zusammen arbeiten. Ein Beispiel für eine Kryptoanalyse, die mehrere Wochen braucht, wäre z. B. das Brechen eines Schlüssels

Plugin	Status Alter Editor	Status Neuer Editor
AES	stabil	getestet
BigNumber	stabil	getestet
BooleanOperators	stabil	getestet
Converter	stabil	getestet
CostFunction	wird getestet	getestet
DES	stabil	getestet
Gate	in Entwicklung	getestet
IncDec	stabil	getestet
KeySearcher	in Entwicklung	getestet
PrimesGenerator	in Entwicklung	getestet
QuadraticSieve	stabil	getestet
RSA	stabil	getestet
SDES	stabil	getestet
TextInput	stabil	getestet
TextOutput	stabil	getestet
Vigenere	getestet	getestet

Tabelle 5.1: Bereits im neuen Editor vollständig getestete Plugins

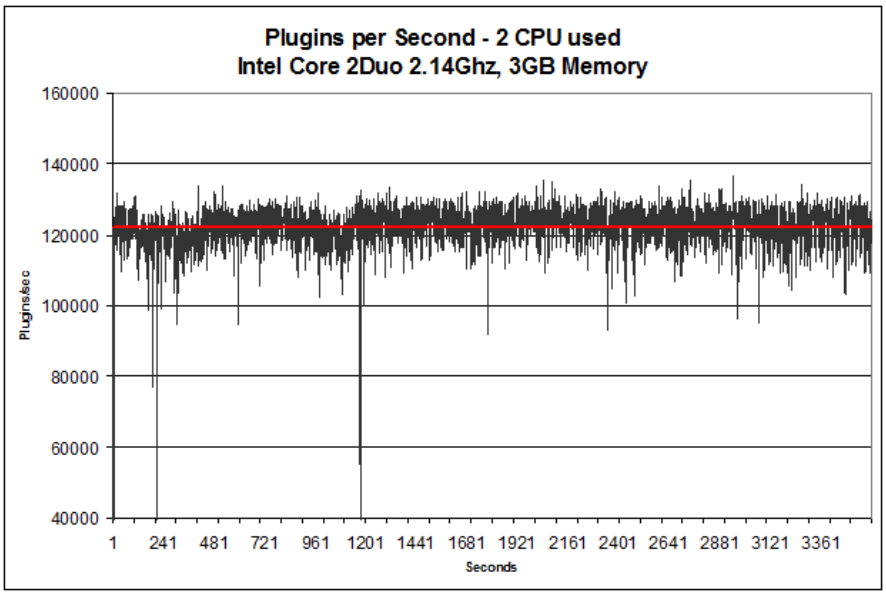

Abbildung 5.2: Geschwindigkeit der Ausführungsmaschine mit zwei Prozessoren

mittels Brute-Force. Dies ermöglicht z. B. das in CT2 enthaltene Keysearcher-Plugin. Das BenchmarkProtocol, das die aktuelle Ausführungsgeschwindigkeit liefert, misst die aktuelle Speichernutzung und gibt diese mittels einer Textdatei aus. Diese wurde genutzt, um die Speichernutzung auszuwerten.

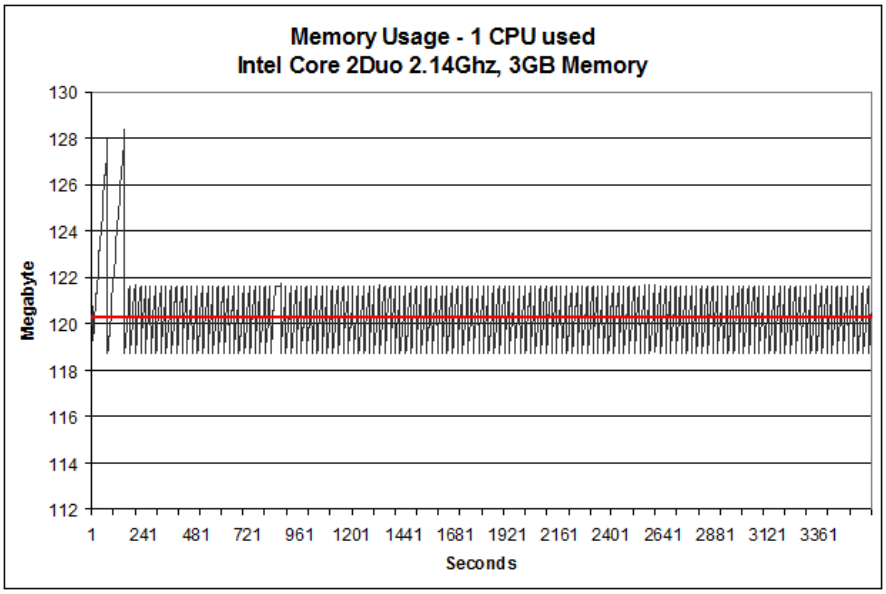

Abbildung 5.3: Speichernutzung der Ausführungsmaschine mit einem Prozessor

In Abbildung 5.4 ist die Größe des, von CT2 und dem neuen Editor genutzten, Arbeitsspeicher, über einen Zeitraum von einer Stunde, zu erkennen. CT2 belegte auf dem Testystem durchschnittlich 120,30 MB. Am zackigen Verlauf der Speicherkurve kann man erkennen, dass kontinuierlich Speicher von der virtuellen Maschine von .Net reserviert wurde. Dieser wurde aber ebenso in regelmäßigen Abständen vom GarbageCollector, der für die Bereinigung von nicht genutzten Speicher zuständig ist, aufgeräumt. Zu erkennen ist außerdem, dass der Speicher immer wieder auf den Ursprungszustand aufgeräumt wurde. Hieraus lässt sich schließen, dass der neue Editor keine Speicherlöcher vorweist. Für zwei genutzte Prozessoren lässt sich ein ähnliches Bild der Speichernutzung in Abbildung 5.4 beobachten. Der genutzte Arbeitsspeicher liegt hier, aufgrund der Ausführung von zwei statt einem Thread, durchschnittlich bei 126,71 MB. Jedoch lässt sich ein ähnlicher Wechsel zwischen Allokation und Freigabe von Arbeitsspeicher erkennen, wie bei der Ausführung mit nur einem Prozessor. Der genutzte Arbeitsspeicher fällt auch hier immer wieder auf seinen Ursprungszustand zurück.

5.3 Auswertung

Mittels der in den beiden vorherigen Unterkapiteln vorgenommen Messungen konnte gezeigt werden, dass die neue Ausführungsmaschine deutlich schneller Pluginketten ausführen kann, als es der alte Editor kann. Im Gegensatz zum alten Editor, der nur einige zehn bis hundert Plugins pro Sekunde ausführen kann, bringt es der neue Editor,

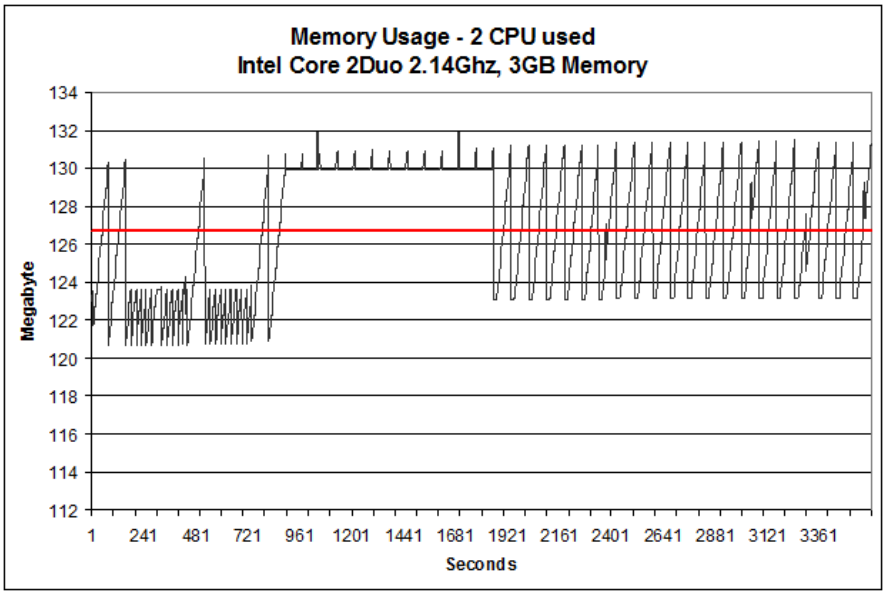

Abbildung 5.4: Speichernutzung der Ausführungsmaschine mit zwei Prozessoren

mit einem Prozessor auf 81.500 Plugins pro Sekunde und mit Nutzung von zwei Prozessoren auf sogar 122.300 Plugins pro Sekunde. Die in der Einleitung geforderte schnellere Ausführungsgeschwindigkeit ist somit erreicht worden.

Durch die Speichernutzungsmessungen konnte außerdem gezeigt werden, dass die neue Ausführungsmaschine auch über längere Zeiträume arbeiten und eine saubere Speichernutzung vorweisen kann.

Mittels des Gears4Net Frameworks und der Orientierung der Entwicklung an Petrinetzen konnte außerdem eine deterministische Ausführungsmaschine konstruiert werden, die automatisch über eine Flusskontrolle verfügt. Ein festes Regelwerk, wann und wie Plugins ausgeführt werden dürfen, und die Belegung der Connectoren, ähnlich den Stellen im Petrinetz, ermöglicht diese Flusskontrolle.

Durch das Einfügen aller Plugins auf den Workspace des neuen Editors konnte gezeigt werden, dass zumindest alle Plugins vom neuen Editor geladen und dargestellt werden können. Beispielhaft wurden einige Plugins dann in Samples eingebaut und es wurde gezeigt, dass der neue Editor mindestens genau so mächtig wie der alte Editor ist.

Der neue Editor verfügt über eine Persistierung des Models, die das Speichern und Laden, wie es in Kapitel 2.1.3 gefordert wurde, umsetzt. PluginModels besitzen eine Größe und Position die einfach geändert werden kann (F03). Sie können mittels des Workspacemodels neu erstellt oder auch gelöscht werden (F02). Mittels des neu entwickelten Models können die Ausgänge und Eingänge von Plugins einfach miteinander mittels ConnectionModels verbunden werden (F04). Die im neuen Editor erstellten Pluginketten können ausgeführt und die Ausführung auch angehalten werden (F07). Sofern ein Plugin eine Repräsentation oder Einstellmöglichkeiten besitzt, werden sie durch das PluginModel verfügbar gemacht (F05,F06). Das Model, das die Datengrundlage der Pluginketten darstellt, kann geladen und gespeichert werden (F08).

Somit sind alle Funktionen, die das Kapitel 2.1.3, innerhalb des neuen Editors und der Ausführungsmaschine umgesetzt worden. Die View, die nicht Teil dieser Bachelorarbeit ist, wurde von Viktor Matkovic erstellt (F01,F02,F03,F05,F06).

6 Ausblick und Zusammenfassung

Nachdem im vorherigen Kapitel durch die Evaluation gezeigt wurde, dass der neue Editor die an ihn gestellten Anforderungen erfüllt, wird in diesem Kapitel ein Ausblick auf die mögliche Weiterentwicklung des Editors und der Ausführungsmaschine gegeben. Gegen Ende dieses Kapitel steht eine kurze Zusammenfassung der Arbeit.

6.1 Ausblick

Die in dieser Arbeit entwickelte Ausführungsmaschine wurde konzeptionell mit Hilfe von Petrinetzen entwickelt und umgesetzt. Die Datenweitergabe zwischen einzelnen Pluginmodels erfolgt in einzelnen Datenstrukturen. So können Zahlen, Byte-Arrays, Strings aber auch komplexere Objekte wie z. B. BigIntegers, die eine beliebig stellige ganze Zahl aufnehmen können, mit Hilfe der ConnectorModels und ConnectionModels von einem PluginModel zum nächsten transportiert werden.

6.1.1 Verarbeitung von Datenströmen

Erst durch die Weitergabe einer der beschriebenen Datenstrukturen und durch die vollständige Belegung aller notwendigen Eingänge darf ein PluginModel bzw. das in ihm enthaltene Plugin ausgeführt werden. Für die Modellierung von Pluginketten die nur mit Datenblöcken arbeiten, ist dieses Vorgehen ausreichend. Neben der Datenübermittelung in Blöcken bietet .Net, wie auch viele andere Plattformen, die Datenübermittelung mittels Datenströmen. Ein Datenstrom ist ein kontinuierlicher Fluss von Daten, deren Ende nicht von vornherein bekannt sein muss. Mit Hilfe solcher Datenströme können große Datensätze in einzelnen kleinen Blöcken transportiert werden. Datenströme werden häufig mittels blockierenden Aufrufen genutzt. So wartet und blockiert eine datenlesende Instanz so lange bis eine datenschreibende Instanz Daten in den Strom schreibt. Innerhalb der Ausführungsmaschine des neuen Editors ist die Verarbeitung solcher Datenströme noch nicht möglich. Das Gears4Net-Framework, das die Grundlage für die Ausführungsmaschine ist, fordert nicht-blockierenden Code innerhalb der Protokollinstanzen. Nur durch dieses kooperative Multitasking können alle Protokollinstanzen Ausführungszeit von den Gears4Net-Schedulern erhalten. Dennoch sind Datenströme notwendig, falls innerhalb von CT2 Daten in Größenordnungen einer CD oder einer DVD transportiert werden sollen. In Zukunft bietet sich eine Weiterentwicklung sogenannter CrypToolStreams oder deren Nachfolger CStreams an, welche die in CT2 implementierten Datenströme sind. Außerdem sollten das Model und die Ausführungsmaschine für den Gebrauch solcher Datenströme weiterentwickelt werden.

6.1.2 Nutzung einer Update-Methode innerhalb des IPlugin

Aktualisierungen von GUI-Elementen werden innerhalb der Plugins durch den Pluginprogrammierer vorgenommen. Sofern der Zustand eines Model-Elements verändert wurde, ermöglicht das Setzen eines Flags (GuiNeedsUpdate) innerhalb des neuen Editors, dass die View während der Ausführung Rechenzeit von der Ausführungsmaschine erhält. Die Ausführungsmaschine beinhaltet einen Mechanismus, der alle Update-Methoden von View-Elementen aufruft, die dieses Flag gesetzt haben. Die Ausführung der Update-Methoden wird automatisch im Kontext des GUI-Threads stattfinden. Somit ist es innerhalb dieser Methoden überhaupt erst möglich auf die WPF-Komponenten der View zuzugreifen, ohne eine Exception auszulösen. Die in CT2 bereits vorhanden Plugins aktualisieren ihre GUI-Elemente auch selbstständig. Sie nutzen allerdings nicht diesen Mechanismus, sondern führen diese Änderungen selbstständig ebenfalls im Kontext des GUI-Threads aus. Jedoch müssen sie bei jeder Aktualisierung immer erneut in diesen Kontext wechseln. Diese Wechsel kosten Rechenzeit und verlangsamen die Ausführung deutlich. Eine Weiterentwicklung wäre die Einführung eines GuiNeedsUpdate-Flag und einer Update-Methode innerhalb des Interfaces IPlugin. So könnten die internen Plugins ebenfalls den Mechanismus nutzen und es würde eine Menge Rechenzeit gespart. Alle, in CT2 vorhandenen, Plugins müssen dafür umgeschrieben werden. Durch die Nutzung des in Kapitel 3 vorgestellten UpdateGuiProtocol würde die Ausführung von vielen Pluginketten erheblich gesteigert werden, da eine große Anzahl von Kontextwechseln so wegfallen würden.

6.1.3 Migration der bestehenden Samples

CT2 verfügt über ein breites Spektrum an fertigen Samples, die bereits vorgefertigte Pluginketten. Diese Samples sollen Einsteigern den Einstieg in CT erleichtern. Innerhalb dieser Bachelorarbeit wurde eine kleine Anzahl von ca. 20 Samples erstellt, die jedoch nur einen kleinen Teil der bereits vorhandenen alten Samples ersetzen könnten. Sobald der neue Editor den alten Editor vollständig ablöst, können die alten Samples im neuen Editor, durch das neue Datenformat bedingt, nicht mehr geladen werden. Alle bereits bestehenden Samples müssten analysiert und innerhalb des neuen Editors nachgebaut und innerhalb der Versionsverwaltung von CT2 gespeichert werden. Durch die Bereitstellung von Text und Bildern auf dem Workspace könnten diese neuen Samples noch einsteigerfreundlicher konstruiert werden. So könnten beschreibende Grafiken und informative Texte die Samples einfacher und verständlicher machen.

6.2 Zusammenfassung

Innerhalb dieser Bachelorarbeit wurden das Model und die Ausführungsmaschine für einen neuen Editor für CT2 entwickelt. Zunächst wurde dargestellt, warum eine solche Neuentwicklung sinnvoll war. Der bereits vorhandene Editor zeigte Schwächen innerhalb der deterministischen Ausführung und bot auch nicht die gewünschte Ausführungsgeschwindigkeit, die jetzt durch den neuen Editor erreicht werden kann. Innerhalb des zweiten Kapitels wurden die grundlegenden Technologien dargestellt, die während der Entwicklung des neuen Editors und der Ausführungsmaschine eingesetzt wurden. Hier

wurde zunächst CT2 vorgestellt. Sowohl der Aufbau der Oberfläche, als auch die Handhabung eines CT2 Editors als auch die Grundlegenden Funktionen, die ein solcher Editor bieten muss, wurden diskutiert. Zu den weiteren Grundlagen gehörte das MVC-Muster, das die Entwicklung von Anwendungen mit Benutzeroberflächen ermöglicht. Diese Anwendungen werden in die drei Teile, das Model, die View und den Controller aufgeteilt. Das Model stellt die Datenbasis, der Controller steuert die Anwendung und die View bietet Sichten auf das Model. Jede einzelne Komponente kann einzeln ausgetauscht werden, und so konnten View und Controller von Viktor Matkovic innerhalb seiner Bachelorarbeit entwickelt werden. Danach wurden Petrinetze beschrieben, die als Konzeptgrundlage dienten. Petrinetze sind Modelle für die Modellierung und das Design von nebenläufigen Anwendungen. Daraufhin wurde Gears4Net vorgestellt, das ein Framework für die Entwicklung von massiv-parallelen Anwendungen bietet. Gegen Ende der Grundlagen wurde noch die Auszeichnungssprache XML vorgestellt, die für die Persistierung des Models genutzt wurde.

Innerhalb des Konzeptes wurden die einzelnen Model-Elemente WorkspaceModel, PluginModel, ConnectorModel und ConnectionModel vorgestellt. Diese bilden die Datengrundlage für die neuen Pluginketten des neuen Editors. Die einzelnen Elemente wurden den Elementen von Petrinetzen nachempfunden. Dabei entspricht das WorkspaceModel der Zeichenfläche, die PluginModels den Transitionen, das ConnectorModel den Stellen und das ConenctionModel den Kanten. Gegen Ende des Konzepts wurde das Konzept für die neue Ausführungsmaschine vorgestellt, die das Model ausführen kann. Diese orientiert sich ebenso an Petrinetzen. Ein PluginModel darf immer genau dann ausgeführt werden, falls alle seine Eingänge belegt und seine Ausgänge frei sind.

Im Kapitel Implementation wurde die Umsetzung des Konzeptes beschrieben. Das Model wurde durch mehrere Klassen implementiert, die namentlich den konzeptionellen Model-Elementen entsprechen. Das WorkspaceModel bietet Methoden, um alle weiteren Model-Elemente zu erstellen. Die Ausführungsmaschine wurde mit Hilfe des Gears4Net-Frameworks entwickelt. Für sie wurde ein eigener, multithreadfähiger Scheduler geschrieben. Die Ausführungsmaschine enthält zudem ein BenchmarkProtocol für Geschwindigkeitsmessungen und Speichermessungen und ein UpdateGuiProtocol, um die View zu aktualisieren.

Durch die Evaluation konnte gezeigt werden, dass die neue Ausführungsmaschine deutlich schneller arbeitet als die Ausführung im alten Editor möglich war. Es wurde auch gezeigt, dass der neue Editor und die neue Ausführungsmaschine über einen längeren Zeitraum stabil arbeiten kann. Hier wurde der Verlauf des reservierten Arbeitsspeichers dargestellt, der relativ konstant bleibt. Es existieren keine Speicherlöcher und allokierter Speicher wird kontinuirlich wieder durch den GarbageCollector freigeben. Innerhalb der Evaluation wurde ebenso gezeigt, dass eine große Anzahl von Plugins getestet und zumindest jedes Plugin innerhalb des Editors geladen werden kann.

Innerhalb dieses Kapitels steht ein Ausblick auf die mögliche Weiterentwicklung des neuen Editors, diese Zusammenfassung und ein abschließendes Fazit.

6.3 Fazit

Durch diese Bachelorarbeit wurde für CT2 ein Teil des neuen Editors entwickelt. Sowohl das Model, als auch die Ausführungsmaschine wurden für eine schnelle Ausführung, während der gesamten Entwicklungszeit, optimiert. Es konnte gezeigt werden, dass eine schnelle Ausführung möglich ist. Durch den neuen Editor erhält CT2 eine modernere Oberfläche und eine schnellere Ausführungsmaschine. Durch die Implementierung eines XMLSerializers ist CT2 um ein einfach erweiterbares Datenformat erweitert worden. Mit Hilfe von Bildern und Texten können Samples nun einsteigerfreundlicher gestaltet werden. Werden die in Kapitel 6.1 Änderungen am neuen Editor umgesetzt, kann der neue Editor in naher Zukunft den alten Editor ablösen.

Literaturverzeichnis

[ERH04] Elliotte Rusty Harold, W. Scott Means: *XML in a nutshell.* O'Reilly Media, Inc., 2004.

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[GB07] Grady Booch, James Rumbaugh, Ivar Jacobson: *Das UML Benutzerhandbuch.* Addison-Wesley, 2007.

[JL08] Jesse Liberty, Donald Xie: *Programmieren mit C# 3.0.* O'Reilly Germany, 2008.

[JW08] Jörg Wegener, Holger Schwichtenberg: *Windows Presentation Foundation - WPF: grafische Benutzerschnittstellen mit .NET 3.5.* Hanser Verlag, 2008.

[LP08] Lutz Priese, Harro Wimmel: *Petri-Netze.* Springer, 2008.

[Mat10] Matkovic, Viktor: *Design und Entwicklung einer erweiterten graphischen Benutzerschnittstelle für CrypTool 2.0*, Bachelorarbeit, Universität Duisburg-Essen, 2010.

[MS09] Martin Saternus, Torben Weis, Sebastian Holzapfel Arno Wacker: *Gears4Net - an Asynchronous Programming Model.* Technischer Bericht, Universität Duisburg-Essen, 2009.

[msd10] msdn: *Einführung in die XML-Serialisierung.* Technischer Bericht, Microsoft Corporation, 2010.

[Sch08] Schmid, Thomas: *Untersuchungen zur visuellen Programmierung: Methodik und Umsetzung in moderner Component Plugin-Architektur auf der .NET-Plattform.* Diplomarbeit, Universität Siegen, 2008.

Versicherung an Eides Statt

Ich, Nils Kopal, Matrikelnummer 2233615, wohnhaft in 47441 Moers, versichere an Eides Statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen übernommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe.

Ich versichere an Eides Statt, dass ich die vorgenannten Angaben nach bestem Wissen und Gewissen gemacht habe und dass die Angaben der Wahrheit entsprechen und ich nichts verschwiegen habe.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 163 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Moers, 28. September 2010 Nils Kopal