Fachbereich Informatik und Medien

MASTERARBEIT

Das Konzept der Brückenknoten
Einbindung einer imperativen Klassenbibliothek in ein Datenflussprogrammiersystem

Vorgelegt von: Tobias Kiertscher

am: 28.05.2010

zum

Erlangen des akademischen Grades

MASTER OF SCIENCE

(M.Sc.)

Erstbetreuer: Prof. Dr. rer. nat. F. Mündemann

Zweitbetreuer: Prof. Dr. sc. techn. H. Loose

Version: 1556

Danksagung

> Gott aber vermag euch jede Gnade überreichlich zu geben, damit ihr in allem allezeit alle Genüge habt und überreich seid zu jedem guten Werk.
>
> *2. Korinther 9, 8*

Mein herzlicher Dank gilt Prof. Dr. Mündemann, der mir durch seine Betreuung ermöglicht hat, das Beste aus meinen Ideen zu machen.

Meiner lieben Frau Susann möchte ich für die unbedingte Unterstützung danken. Sie hat mich mit meinen Hochs und Tiefs ertragen, mir immer wieder Mut gemacht und mich tatkräftig durch Korrekturlesen unterstützt.

Auch meine Kollegin Katja Orlowski darf nicht unerwähnt bleiben. Sie hat mir beim Korrekturlesen der Arbeit ebenfalls sehr geholfen.

Ein besonderer Dank geht an den gesamten Fachbereich Informatik und Medien, der mir eine einmalige Plattform für meine Arbeit zur Verfügung gestellt hat.

Danke für die viele Unterstützung!

Zusammenfassung

Neben einer großen Anzahl an textbasierten Programmiersprachen, von denen einige eine sehr weite Verbreitung gefunden haben, existiert eine Vielzahl von visuellen, datenflussorientierten Programmiersystemen. Diese Systeme konzentrieren sich in der Regel auf eine begrenzte Anwendungsdomäne und haben sich bisher nicht im Sinne einer Allzwecksprache für die allgemeine Anwendungsentwicklung durchsetzen können.

Die Entwicklungsumgebung DynamicNodes, welche zur Klasse der visuellen Datenflussprogrammiersysteme gehört, besitzt nur eine kleine Anzahl an Operationsknoten die sich auf den Anwendungsbereich Bildverarbeitung konzentrieren. DynamicNodes basiert auf der Microsoft .NET-Technologie, für die umfangreiche Klassenbibliotheken für vielfältige Anwendungsbereiche verfügbar sind. Diese Klassenbibliotheken können jedoch in DynamicNodes nicht direkt genutzt werden.

Diese Arbeit stellt das Konzept der Brückenknoten vor, mit dem imperative, objektorientierte Klassenbibliotheken in ein datenflussorientiertes Programmiersystem eingebunden werden können. Damit wird es möglich, das Potential von existierenden Klassenbibliotheken einer weit verbreiteten Programmiersprache, wie Java oder C#, direkt in einem datenflussorientierten Programmiersystem zu nutzen und dessen Anwendungsgebiet stark zu erweitern.

Des Weiteren zeigt diese Arbeit die Einbindung einer imperativen, objektorientierten Klassenbibliothek in ein datenflussorientiertes Programmiersystem exemplarisch mit dem Entwurf und der Implementierung von Brückenknoten, welche die .NET-Klassenbibliotheken in das Datenflussprogrammiersystem DynamicNodes einbinden.

Abstract

Besides a large number of textual programming languages, some of which have found a wide distribution, there is a variety of visual data-flow programming systems. These are usually focused on a limited application domain and have not been able to prevail in terms of a general-purpose language for general application development.

The development environment DynamicNodes, which belongs to the class of visual data-flow programming systems, has only a small number of operational nodes that focus on image processing. DynamicNodes is based on Microsoft .NET technology. For .NET rich class libraries for various application domains are available. These class libraries can not be used directly in DynamicNodes.

This thesis introduces the concept of bridge nodes, which allows to bind imperative, object-oriented class libraries in a data-flow programming system. This makes it possible to use the potential of existing class libraries from popular imperative, object-oriented programming languages, such as Java or C#, directly in a data-flow programming system and extends its scope of application considerably.

Further this thesis gives an example for binding an imperativ, object-oriented class library to a data-flow programming system by the design and implementation of bridge nodes, which bind the .NET class libraries to the data-flow programming system DynamicNodes.

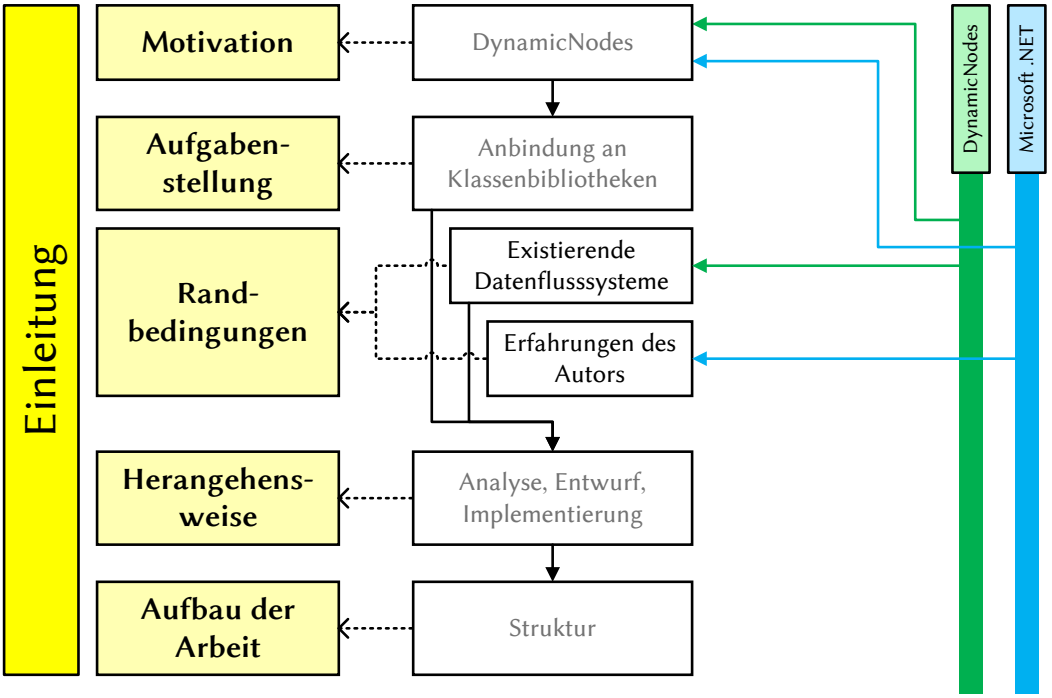
Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1 Einleitung



Das erste Kapitel dieser Arbeit bildet die Einleitung und gliedert sich in fünf kurze Abschnitte. In 1.1 wird die Motivation für die Arbeit beschrieben. Anschließend wird in 1.2 die Aufgabenstellung ausformuliert und in 1.3 werden einige Randbedingungen erläutert. Die Herangehensweise für die Bearbeitung der Aufgabenstellung wird in 1.4 dargestellt und zum Abschluss des Kapitels wird in 1.5 der Aufbau der Arbeit beschrieben.

1.1 Motivation

Der Autor hat im Rahmen seiner Diplomarbeit das Datenflussprogrammiersystem DynamicNodes entwickelt (vgl. [Kier07]). Dieses System wurde auf der Basis des .NET-Frameworks implementiert. Die Leistungsfähigkeit eines Datenflussprogrammiersystems hängt im Wesentlichen von der Verfügbarkeit von geeigneten Operationsknoten ab. Da der Autor nach Fertigstellung der Diplomarbeit nur noch wenig Zeit für die Weiterentwicklung von DynamicNodes aufbringen konnte, blieb die Anzahl der verfügbaren Operationsknoten für DynamicNodes recht bescheiden. Eine ernsthafte Programmierung mit dem System war so nicht möglich.

Die Idee, die zu dieser Arbeit führte, war die direkte Einbindung der .NET-Klassenbibliotheken in die datenflussorientierte Programmierung. Die .NET-Klassenbibliotheken stellen eine große Anzahl von gut getesteten Programmbausteinen für die unterschiedlichsten Anwendungsbereiche zur Verfügung. Diese Programmbausteine sollen in Form von Operationsknoten in DynamicNodes verfügbar gemacht werden. Diese Arbeit versucht deshalb, eine Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken zu entwerfen. Eine solche Brücke verspricht, die Leistungsfähigkeit von DynamicNodes wesentlich zu steigern.

1.2 Aufgabenstellung

Aufgabe der Arbeit ist es, eine Möglichkeit zu finden, eine imperative, objektorientierte Klassenbibliothek derart in ein Datenflussprogrammiersystem einzubinden, dass die Datenstrukturen und Algorithmen der Klassenbibliothek, mit Hilfe von Operationsknoten, in einem Datenflussgraphen genutzt werden können. Da sich ein Datenflussgraph sowohl in der Programmstruktur als auch in der Kontrolle des Programmablaufs grundlegend von einem imperativen, objektorientierten Programm unterscheidet, muss eine Brücke zwischen der datenflussorientierten und der imperativen, objektorientierten Programmierung geschlagen werden.

In dieser Arbeit sollen zunächst allgemein verwendbare Lösungsansätze für die Einbindung von imperativen, objektorientierten Klassenbibliotheken in Datenflussprogrammiersysteme

entwickelt werden. Anschließend soll ihre Tauglichkeit durch den Entwurf und die Implementierung einer Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken unter Beweis gestellt werden. Für den Entwurf sind Entwurfsziele zu definieren, deren Erreichen durch die Implementierung zu bewerten ist.

1.3 Randbedingungen

In diesem Abschnitt werden kurz die Randbedingungen geschildert, unter denen diese Arbeit entstanden ist.

1.3.1 Existierende datenflussorientierte Programmiersysteme

Zu den Randbedingungen der Arbeit gehören bereits existierende datenflussorientierte Programmiersysteme. Denn die zu erarbeitenden allgemeinen Lösungsansätze sollen für eine möglichst große Anzahl von Kombinationen aus datenflussorientierten Programmiersystemen und imperativen, objektorientierten Klassenbibliotheken nutzbar sein.

DynamicNodes ist ein softwarebasiertes, visuelles Programmiersystem. Um den Rahmen dieser Arbeit nicht zu sprengen, soll an dieser Stelle nur eine Auswahl von softwarebasierten, datenflussorientierten Programmiersystemen genannt werden, die über eine visuelle Programmierschnittstelle verfügen.

- *DynamicNodes* [Kier09a]
  Ein noch experimentelles, visuelles Programmiersystem auf der Basis des Microsoft .NET Frameworks. Entwickelt von Tobias Kiertscher, konzentriert sich das System bisher auf einfache Aufgaben im Bereich Bildverarbeitung und ist gut geeignet für den Einsatz in der Lehre.

- *LabVIEW* [Nati10]
  Eine Entwicklungsumgebung, die auf hardwarenahe Anwendungen, wie Maschinensteuerung und Regelsysteme, spezialisiert ist, sich aber auch als allgemeine Mehrzweck-Programmierplattform anbietet. LabVIEW wird von National Instruments entwickelt.

- *Microsoft Visual Programming Language* [Micr09]
  Eine visuelle Programmiersprache von Microsoft, deren Entwicklung eng mit dem Robotics Developer Studio verzahnt ist. Aus diesem Grund liegt der Fokus dieses Systems auch auf der Programmierung von mobilen Systemen.

- *Microsoft Windows Workflow Foundation* [Espo]
  Die Workflow Foundation ist ein Teil des .NET-Frameworks und wird innerhalb von VisualStudio durch einen grafischen Designer unterstützt. Von Micrsoft entwickelt,

liegt der Schwerpunkt dieses Systems auf der Prozessmodellierung und der Entwicklung von prozessorientierten WebServices.

- *OpenWire* [Mito10]
  Die OpenSource Entwicklungsumgebung OpenWire wird von der Firma Mitov Software gepflegt und bildet die Grundlage für deren Produktpalette im Bereich der Audio-, Bild- und Signalverarbeitung bzw. -analyse.

- *Simulink* [Math10]
  Die Firma MathWorks bietet Simulink als ergänzendes Produkt zu MATLAB an. Simulink ermöglicht das Komponieren von Anwendungsblöcken auf der Basis der Entwicklungsumgebung MATLAB. Der Fokus dieses Systems liegt auf wissenschaftlichen Anwendungen. Der Funktionsumfang der verschiedenen für Simulink angebotenen Pakete ist aber thematisch sehr breit gefächert.

- *Tersus* [Ters10]
  Die visuelle Entwicklungsumgebung Tersus Studio basiert auf Eclipse und konzentriert sich auf die Webentwicklung im Java-Umfeld. Die Tersus Software Ltd., welche hinter dem Programmiersystem steht, bewirbt das Produkt mit dem Versprechen, dass vollständig visuell entwickelt werden kann und kein Quellcode bearbeitet werden muss.

- *vvvv* [vvvv10]
  Das Mehrzweck-Programmiersystem vvvv ist ein in C++ entwickeltes und von der Entwicklergruppe um Joreg, Wolf, Gregor und Oschatz gepflegtes Entwicklungswerkzeug. Der Schwerpunkt des Systems liegt auf der Erzeugung und Verarbeitung von Multimedia-Datenströmen (Video, 3D, Audio, etc.) in Echtzeit.

1.3.2 Erfahrungen des Autors

Der Autor programmiert seit dem Jahr 1999 in verschiedenen Programmiersprachen, u.a. BASIC, VisualBasic, C, C++, Java, C#, F#, MATLAB, PowerShell und Python. Während seines Diplomstudiums begann er das visuelle Programmiersystem *DynamicNodes* mit einem Schwerpunkt auf Bildverarbeitung zu entwickeln, welches auch Gegenstand seiner Diplomarbeit im Jahr 2007 war (vgl. [Kier07]). Bei der Entwicklung von DynamicNodes setzt der Autor die Entwicklungsumgebung Microsoft Visual Studio und die Programmiersprache C# ein.

In den beiden folgenden Jahren hat der Autor sich weiter in das Thema der Datenflussprogrammierung und der visuellen Programmierung eingearbeitet (vgl. [Kier09b]) und versucht, innovative Ansätze für die Weiterentwicklung von DynamicNodes im Speziellen und die visuelle Programmierung im Allgemeinen zu finden. Die vorliegende Masterarbeit ist als Teilergebnis dieser Bemühungen zu betrachten.

1.4 Herangehensweise

Für die Erfüllung der Aufgabenstellung in *Abschnitt 1.2 Aufgabenstellung* müssen Daten zwischen Datenflussprogrammen und imperativen, objektorientierten Programmen ausgetauscht werden. Des Weiteren muss die Programmablaufkontrolle der beiden Programmierparadigmen in Einklang gebracht werden. Dazu ist die Analyse beider Programmierparadigmen notwendig. Ausgehend von der Analyse sollen grundlegende Konzepte erarbeitet werden, welche die Anbindung von Datenflussgraphen an Fremdsysteme ermöglichen. Diese Konzepte sind anschließend für die Einbindung von imperativen, objektorientierten Klassenbibliotheken zu spezialisieren.

Um der Verschiedenartigkeit von existierenden Datenflussprogrammiersystemen und Klassenbibliotheken gerecht zu werden, soll eine Reihe von Lösungsansätzen entwickelt werden, die für unterschiedliche Szenarien geeignet sind. Es bietet sich an dieser Stelle an, einige fiktive Beispielszenarien zu entwerfen und die Auswahl eines jeweils passenden Lösungsansatzes zu beschreiben. Dabei werden die Lösungsansätze in einem realitätsnahen Umfeld betrachtet. Für den Entwurf einer Brücke, für eine konkrete Kombination aus Datenflussprogrammiersystem und imperativer, objektorientierter Klassenbibliothek, wird dadurch eine kleine Hilfestellung gegeben.

Ist das Konzept einer Brücke zwischen einem Datenflussprogrammiersystem und imperativen, objektorientierten Klassenbibliotheken im Allgemeinen behandelt, soll eine konkrete Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken entworfen werden. Bei dem Entwurf soll auf die im Allgemeinen entwickelten Lösungsansätze zurückgegriffen werden. Die entworfene Brücke soll anschließend implementiert und bewertet werden.

1.5 Aufbau der Arbeit

Die Arbeit gliedert sich in sieben Kapitel. Ein Überblick über den Gesamtaufbau gibt *Abbildung 1.1.*

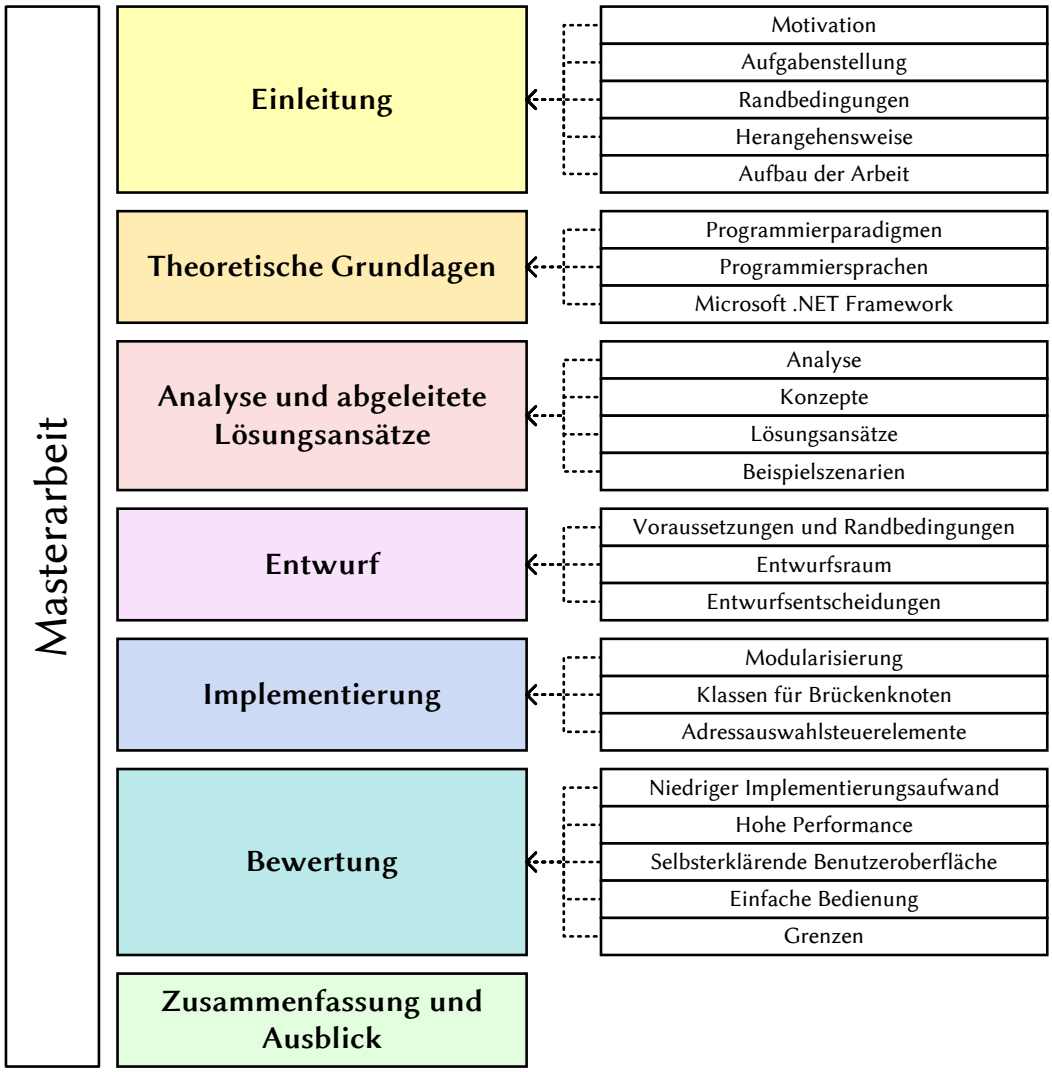


Abbildung 1.1: Aufbau der Arbeit

Am Anfang jedes Kapitels steht eine Grafik zur Veranschlaulichung der Kapitelstruktur.

Das erste Kapitel – *Kapitel 1 Einleitung* – enthält die folgenden Abschnitte:

- **Motivation** ab Seite 2
- **Aufgabenstellung** ab Seite 2
- **Randbedingungen** ab Seite 3
- **Herangehensweise** ab Seite 5

- **Aufbau der Arbeit** ab Seite 6

Das zweite Kapitel – *Kapitel 2 Theoretische Grundlagen* – enthält die folgenden Abschnitte:

- **Programmierparadigmen** ab Seite 10
- **Programmiersprachen** ab Seite 20
- **Microsoft .NET Framework** ab Seite 20

Das dritte Kapitel – *Kapitel 3 Analyse und abgeleitete Lösungsansätze* – enthält die folgenden Abschnitte:

- **Analyse** ab Seite 30
- **Konzepte** ab Seite 34
- **Lösungsansätze** ab Seite 39
- **Beispielszenarien** ab Seite 46

Das vierte Kapitel – *Kapitel 4 Entwurf*– enthält die folgenden Abschnitte:

- **Voraussetzungen und Randbedingungen** ab Seite 51
- **Entwurfsraum** ab Seite 57
- **Entwurfsentscheidungen** ab Seite 60

Das fünfte Kapitel – *Kapitel 5 Implementierung* – enthält die folgenden Abschnitte:

- **Modularisierung** ab Seite 86
- **Klassen für Brückenknoten** ab Seite 87
- **Adressauswahlsteuerelemente** ab Seite 108

Das sechste Kapitel – *Kapitel 6 Bewertung* – enthält die folgenden Abschnitte:

- **Niedriger Implementierungsaufwand** ab Seite 113
- **Hohe Performance** ab Seite 114
- **Selbsterklärende Benutzeroberfläche** ab Seite 117
- **Einfache Bedienung** ab Seite 119
- **Grenzen** ab Seite 120

Das siebente Kapitel – *Kapitel 7 Zusammenfassung und Ausblick* – ist nicht weiter gegliedert. In diesem Kapitel wird der Inhalt der Arbeit zusammengefasst und es wird ein Ausblick für Verbesserungen und auf nächste Schritte gegeben.

Die Arbeit besitzt einen Anhang – *Kapitel A Anhang* – mit den folgenden Abschnitten:

- **Umfang des Microsoft .NET-Frameworks** ab Seite 137

- **Benchmark für Bindungstechniken** ab Seite 145
- **Diagramme** ab Seite 151
- **Quelltexte** ab Seite 162

In dieser Arbeit werden die folgenden unterschiedlichen Auszeichnungen verwendet:

- *So werden Wörter ausgezeichnet, die eine besondere Bedeutung besitzen oder als neuer Begriff eingeführt werden. Auch Querverweise werden in dieser Form ausgezeichnet.*
- `Dateinamen und Quellcode-Schnippsel werden mit einer monotypen Schrift gesetzt.`
- **Ist ein Wort in einer Aufzählung besonders wichtig, wird es in dieser Form dargestellt**.

Kurze Quelltexte werden in der folgenden Form ausgezeichnet:

```
Dies ist ein kurzer
Beispielquelltext.
```

Ist ein Quelltext etwas länger, wird er in der folgenden Form ausgezeichnet:

```
1 Dies ist ein etwas
2 längerer Quelltext.
```

Im Anhang werden Quelltexte wie folgt gesetzt:

```
1 Dies ist ein Quelltext
2 der im Anhang aufgeführt wird.
```

An vielen Stellen dieser Arbeit wird auf die Dokumentation der .NET-Klassenbibliothek und auf die Dokumentation der Programmierschnittstelle von DynamicNodes Bezug genommen. Die direkten Adressen der Dokumentation werden bei der ersten Verwendung einer Klasse oder eines Klassenmitglieds als Fußnote angegeben. Z. B. in dieser Form: `DynamicNode.Core.INode`[1]. Liegt die Arbeit als Portable Document File (PDF) vor, sind alle URLs aktive Links und können mit einem Klick direkt im Browser aufgerufen werden.

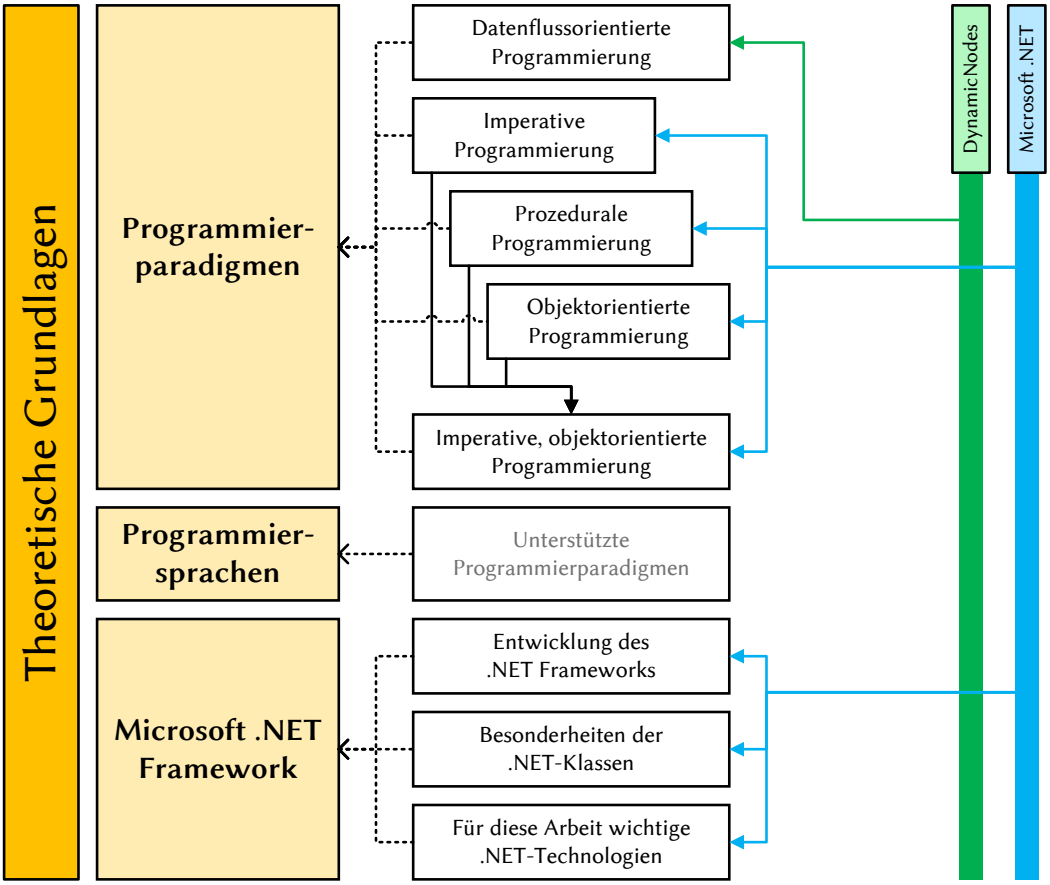
Unter den folgenden URLs sind die Startseiten der Dokumentationen direkt erreichbar.

Microsoft .NET `http://msdn.microsoft.com/en-us/library/ff361664.aspx`

DynamicNodes `http://dynamicnodes.mastersign.de/docs/api/`

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_INode.htm`

2 Theoretische Grundlagen



Dieses Kapitel liefert die theoretischen Grundlagen für die Arbeit und gliedert sich in drei Abschnitte. In dieser Arbeit sollen zwei Programmierparadigmen miteinander verknüpft werden: die datenflussorientierte Programmierung und die imperative, objektorientierte Programmierung. Deshalb wird in Abschnitt 2.1 ein grober Überblick über diese Programmierparadigmen gegeben. Die imperative, objektorientierte Programmierung ist eine Mischform und wird durch die imperative, die prozedurale und die objektorientierte Programmierung beeinflusst. Alle drei Programmierparadigmen werden zunächst unabhängig voneinander vorgestellt. Anschließend wird die Mischform erläutert.

Um die Programmierparadigmen in die aktuelle Softwaretechnik einzuordnen, werden in Abschnitt 2.2 einige zur Zeit weit verbreitete oder vom Autor verwendete Programmiersprachen aufgezählt und deren unterstützte Programmierparadigmen verglichen.

Als wichtige Randbedingung für den praktischen Teil der Arbeit – Entwurf und Implementierung – wird in Abschnitt 2.3 das Microsoft .NET-Framework vorgestellt.

2.1 Programmierparadigmen

Der Begriff Programmierparadigma lässt sich wie folgt definieren:

> Wenn man von Paradigmen in der Programmierung spricht, handelt es sich in der Regel um ein gedankliches Modell mit einem theoretischen Fundament, das es ermöglicht, das Lösen von Problemen auf den Computer zu übertragen. Ein Programmierparadigma ist somit die Grundlage einer Programmiersprache.

Um einen Computer mit der Lösung einer Aufgabe zu betrauen, muss dem Computer einerseits das Problem und andererseits eine Lösungsstrategie übergeben werden. Dies ist in einer Form notwendig, die der Computer verarbeiten kann. Die verschiedenen Paradigmen in der Programmierung geben einen Rahmen für die Übersetzung einer für den Menschen leicht zu verstehenden Sprache in Maschinensprache. Typische Programmierparadigmen sind logisches (vgl. [Lloy87]), funktionales (vgl. [BiWa88]), imperatives (vgl. [HuAi04, S. 43ff]) und objektorientiertes Programmieren (vgl. [Booc91, Booc94]).

In diesem Abschnitt werden die datenflussorientierte, die imperative, die prozedurale und die objektorientierte Programmierung vorgestellt. Anschließend wird auf die in vielen Programmiersprachen verbreitete Mischung aus imperativer, prozeduraler und objektorientierter Programmierung eingegangen.

2.1.1 Datenflussorientierte Programmierung

Das Programmierparadigma *Datenfluss* lässt sich aus zwei primären Perspektiven betrachten. Die Erste ist die strukturelle Sicht. Aus dieser Perspektive wird die Organisation von Daten

und Programm in Beziehung zueinander betrachtet. Die zweite Sicht ist das Berechnungs- oder Ausführungsmodell.

Das Ergebnis der datenflussorientierten Programmierung ist ein *Datenflussprogramm*. Datenflussprogramme benötigen eine Form der *Laufzeitumgebung*. Diese kann aus Hardware oder aus Software bestehen. Der Begriff *Datenflusssystem* bezeichnet ein System aus Laufzeitumgebung und Datenflussprogrammen. Ein System aus Programmierumgebung, Laufzeitumgebung und Datenflussprogrammen wird *Datenflussprogrammiersystem* genannt.

Struktur von Datenflussprogrammen

Die Struktur eines Datenflussprogramms wird durch einen gerichteten Graphen repräsentiert. Dieser sog. *Datenflussgraph* besteht aus Knoten und gerichteten *Kanten*. Die Kanten verbinden die Knoten miteinander. Jeder Knoten repräsentiert einen Verarbeitungsschritt bzw. eine *Operation* und jede Kante einen Transportweg für Daten (vgl. *Abbildung 2.1*).

Wenn ein Knoten mit seiner Arbeit beginnt, nimmt er die Daten von den einlaufenden Kanten als Operatoren entgegen. Nach Abschluss der Operation gibt er die Ergebnisdaten über auslaufende Kanten an nachfolgende Knoten weiter.

Knoten, die ausschließlich auslaufende Kanten besitzen, werden *Quellen* genannt und Knoten, welche ausschließlich einlaufende Kanten besitzen, werden *Senken* genannt.

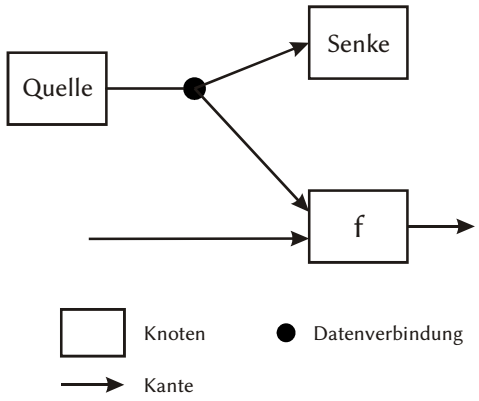


Abbildung 2.1: Datenflussgraph nach [Denn74]

Auf ein Datenflussprogramm als gerichteten Graphen lassen sich Erkenntnisse und Algorithmen aus der Graphentheorie – einem Teilgebiet der Mathematik – anwenden. Eine Einführung zum Thema Graphentheorie bietet bspw. [Volk96].

Einige Datenflusssysteme (z. B. vvvv oder DynamicNodes) verwenden an ihren Knoten definierte Ein- und Ausgänge (vgl. *Abbildung 2.2*). Diese *Anschlüsse* bilden die Schnittstellen eines Knotens.

Für die Nutzung des Datenflussgraphen in der Laufzeitumgebung gibt es zwei unterschiedliche Strategien. Die eine wird *statisch* und die andere *dynamisch* genannt. (Nicht zu verwechseln mit statischen und dynamischen Klassenmitgliedern der objektorientierten Programmierung.)

Eine Laufzeitumgebung, die statisch arbeitet (*static dataflow computing*), erzeugt beim Laden des Datenflussgraphen für jeden Knoten genau eine *Knoteninstanz* (vgl. *[Sh+92, S. 23-24]*).

Eine Laufzeitumgebung, die dynamisch arbeitet (*dynamic dataflow computing*), verwendet den Datenflussgraphen als Grundlage für den Datentransport, erzeugt jedoch eine Knoteninstanz erst in dem Moment, in dem der Knoten aktiviert werden soll (vgl. [Sh+92, S. 24-26]). Bei der dynamischen Ausführung können von einem Knoten zur gleichen Zeit mehrere Knoteninstanzen existieren.

In einem Datenflussgraphen kann es mehrere Knoten geben, welche die gleiche Operation ausführen. Eine Klasse aller Knoten, die die gleiche Operation ausführen, wird *Knotentyp* genannt. Verwendet ein Datenflusssystem definierte Ein- und Ausgänge, werden diese durch den Knotentyp vorgegeben.

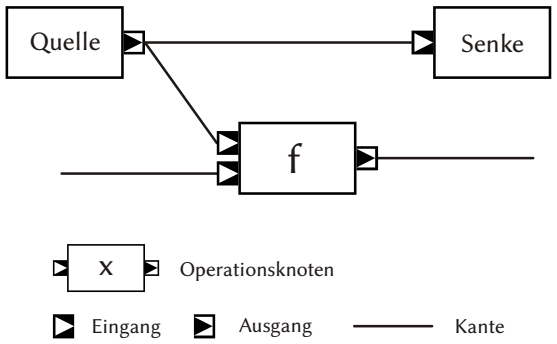


Abbildung 2.2: Datenflussgraph mit Ein- und Ausgängen an den Knoten nach [Kier08, S. 22]

Ausführungsmodell von Datenflussprogrammen

Eine zweite Perspektive der datenflussorientierten Programmierung bildet das Ausführungsmodell. Aus dieser Perspektive wird nicht die Struktur eines Datenflussprogramms betrachtet, sondern die Art und Weise der Ausführung eines Datenflussprogramms.

In der Theorie arbeitet jeder Knoten in einem eigenen Ausführungspfad (*Thread*). Wie in *Struktur von Datenflussprogrammen* auf Seite 11 erwähnt, kann ein Knoten nur dann arbeiten, wenn die notwendigen Daten über die Eingänge eingetroffen sind. Ist das nicht der Fall, wartet der Knoten bis die Daten eintreffen. In einem Datenflussprogramm können alle Knoten gleichzeitig arbeiten, die nicht voneinander abhängig sind, bzw. alle Knoten, welche die

notwendigen Daten an ihren Eingängen besitzen.

In [Sh+92, S. 8-11] werden verschiedene Vorbedingungen für die Aktivierung eines Knotens dargestellt. Einige Ansätze (vgl. [Denn74, Rumb77] fordern, dass an allen einlaufenden Kanten eines Knotens Daten liegen müssen, damit der Knoten arbeiten kann. Andere erlauben oder fordern Operationen, welche mit unvollständigen Eingabemengen auskommen (vgl. [Adam70, Kosi73]).

Da die Synchronisation der Ausführungspfade implizit durch die Daten erfolgt (*data driven*) kommt ein Programmierer bei der Entwicklung von nebenläufigen Programmen i. d. R. ohne explizite Synchronisationsmechanismen wie Sperren, Monitore oder Semaphoren aus.

Eine gute Einbettung des Datenflussparadigmas in den Kontext anderer Paradigmen ist in [RoHa03] zu finden.

Formen von Datenflusssystemen

Datenflusssysteme gibt es in sehr unterschiedlichen Formen, sowohl was das konkrete Konzept in Struktur und Ausführungsmodell betrifft (vgl. [Kier09b]) als auch in Bezug auf die Implementierung. In den 80er-Jahren wurde intensiv im Bereich der Datenflusssysteme als Hardware-Plattform geforscht. Aus dieser Zeit stammen viele bedeutende Arbeiten, welche die heutige Datenflusstheorie stark geprägt haben. Dazu gehören unter anderem [Adam70], [Kosi73], [Denn74], [Rumb77] und [TrBrHo82].

Einen guten Einstieg in die Datenflusstheorie bietet auch [Sh+92]. In diesem Werk werden verschiedene Notationen und Ausführungsmodelle vorgestellt.

In der aktuellen Entwicklung findet die Datenflusstheorie zum einen in der Entwicklung von grafischen Programmiersystemen Anwendung und zum anderen wird versucht, die Vorteile der Datenflusssysteme im Bereich der Programmierung von parallelen Abläufen in moderne Multiparadigmen-Programmiersprachen einfließen zu lassen.

Die Implementierung von Datenflusssystemen auf Hardwareebene konnte sich in den letzten Jahren nicht gegen die Rechnerarchitekturen nach dem Von-Neumann-Prinzip durchsetzen (vgl. [Neum93],[Wa+95, S. 6ff]). Parallelität, als großes Ziel, wird jedoch weiterhin sowohl in der Hardware auf der Suboperationsebene (*Pipeline*: vgl. [Gilo93, S. 80-82]) als auch in der Software auf Block- bis Programmebene weiterentwickelt (vgl. [Inte10, Micr10c]). Die Schnittstelle zwischen Hard- und Software bildet aber in der Mehrzahl der eingesetzten Systeme, im Server- und auch im Arbeitsplatzrechner, das Von-Neumann-Prinzip – implementiert durch x86-kompatible oder verwandte Systeme. Die Klassifikation der Parallelitätsebenen erfolgt an dieser Stelle nach [Unge97, S. 8-10].

Ist heute von Datenflusssystemen die Rede, sind i. d. R. Softwaresysteme gemeint, welche die

datenflussorientierte Programmierung unterstützen, die Programme jedoch auf PC-kompatibler Hardware ausführen.

Diese Entwicklung wird noch dadurch bestärkt, dass sich die aktuelle Hardware von einer reinen Von-Neumann-Architektur immer stärker entfernt. Die aktuellen PC- und Server-Systeme sind auf Hardware-Ebene meist NUMA-Architekturen (vgl. [Unge97, S. 6],[Torp08]) – auch wenn die mehrstufigen Cache-Systeme mittels Cache-Kohärenz-Algorithmen aus Perspektive des Programmierers weitgehend transparent bleiben – und zunehmend auch Mehrprozessormaschinen. Das Datenflussparadigma erleichtert in diesem Kontext die Entwicklung von Programmen mit mehreren Ausführungspfaden.

Kontrollfluss in Datenflussgraphen

Die Kontrolle über die Reihenfolge für die Ausführung der verschiedenen Knoten erfolgt in einem Datenflussprogramm implizit über die Kanten und die Existenz von erforderlichen Daten. Reicht dieser Kontrollmechanismus nicht aus, werden in Datenflusssystemen dedizierte Kontrollkanten eingeführt, welche Kontrollmarken transportieren ([Sh+92, S. 28],[TrBrHo82]). Ist in dieser Arbeit von *Kontrollfluss* die Rede, sind diese Arten von Kontrollkanten gemeint.

Besitzt ein Knoten eine einlaufende Kontrollkante, darf er erst mit der Arbeit beginnen, wenn über die Kontrollkante eine Marke einläuft. Ob in diesem Fall auf den übrigen einlaufenden Kanten Daten vorhanden sein müssen, ist von System zu System verschieden.

Abbildung 2.3 zeigt einen Datenflussgraphen mit Kontrollkanten.

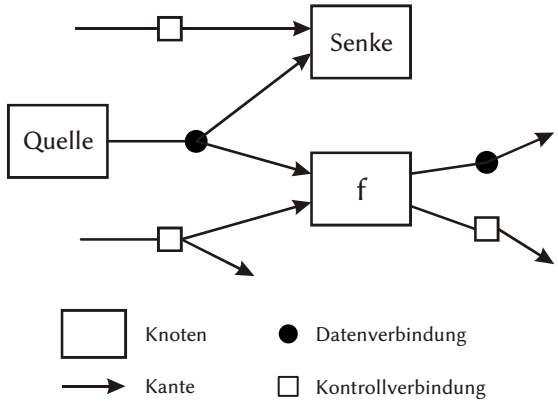


Abbildung 2.3: Datenflussgraph mit Kontrollfluss in Anlehnung an [Denn74, TrBrHo82]

2.1.2 Imperative Programmierung

Eine gute Übersicht über das imperative Programmierparadigma ist in [HuAi04, S. 43ff] zu finden. Da Datenflussprogrammiersysteme implizit parallel arbeiten und die imperative Pro-

grammierung in dieser Arbeit mit der Datenflussprogrammierung verbunden werden soll, wird in diesem Abschnitt sowohl kurz auf die sequentielle, imperative Programmierung als auch auf die parallele, imperative Programmierung eingegangen.

Sequentielle Programmierung

Die imperative Programmierung ist an die Arbeitsweise des von Von-Neumann-Rechners (vgl. [Neum93]) angelehnt, nach dessen Prinzip die IBM-kompatiblen x86- und die aktuell verbreiteten x64-Rechner funktionieren. Ein Von-Neumann-Rechner arbeitet einzelne Befehle in einem Datenkontext ab. Das heißt, die Befehle beziehen sich jeweils auf den aktuellen Zustand des Speichers und können diesen verändern. Das steuernde Element dieses Paradigmas ist ein Befehlszähler, der auf den jeweils aktuell auszuführenden Befehl zeigt und bei Ausführung des Befehls automatisch inkrementiert wird. Durch aktives Ändern des Befehlszählers können Sprünge im Befehlscode vorgenommen werden.

Imperative Programmiersprachen bieten einen Befehlssatz, der i. d. R. abstrakter ist als der Maschinenbefehlssatz. Aber die Arbeitsweise und die wesentlichen Steuerbefehle zur Manipulation des Befehlszählers (wie z. B. bedingter und unbedingter Sprung) entsprechen direkt der Maschinenarbeitsweise. Die heute, in modernen imperativen Programmiersprachen wie Java und C#, üblichen Steuerbefehle `if`, `for`, `while`, `switch`, `break`, `continue` und der Methodenaufruf sind eine Abstraktion für die oben genannten Maschinenbefehle. Sie ermöglichen jedoch nichts anderes, als die Beschreibung einer sequentiellen Verarbeitungsvorschrift, die dem Prozessor genau sagt, wie er was und in welcher Reihenfolge zu tun hat.

Parallele Programmierung

Auf der Ebene der modernen imperativen Programmiersprachen wird ein Großteil der Datenorganisation und programminternen Kommunikation durch Variablen realisiert. Diese können gelesen und geschrieben werden. Arbeitet das Programm mit einem einzelnen Ausführungspfad (*Thread*), ist das ein einfach zu verstehendes Konzept. Arbeiten jedoch mehrere Ausführungspfade in einem Programm gleichzeitig, wird der gemeinsame Speicher, der durch die Variablen zugänglich gemacht wird, schnell zu einem Problem (vgl. [Unge97, S. 54ff],[RaRü07]).

Da die aktuelle Entwicklung der Computer stark in Richtung von Mehrprozessormaschinen läuft (vgl. [Merr08]), muss die Softwaretechnik ausdrucksstarke und leicht zu erlernende Konzepte bereitstellen, um die parallele Programmierung von Mehrprozessorsystemen zu ermöglichen.

In der aktuellen Softwareentwicklung werden zunehmend abstraktere Konzepte und Programmierbibliotheken entwickelt, um den Umgang mit den elementaren Mitteln für die kon-

kurrierende Programmierung (Thread, Semaphore, Monitor, Sperre, u. a.) zu vereinfachen oder sogar vollständig zu verbergen. Ziel dabei ist es, problematische Situationen wie z. B. Deadlocks oder Race Conditions zu verhindern und dennoch das Potential von nebenläufigen Programmen zu eröffnen (vgl. [RaRü07, Unge97]).

2.1.3 Prozedurale Programmierung

Die prozedurale Programmierung ist unter anderem das Paradigma der Programmiersprache C, welche in den letzten Jahrzehnten eine sehr weite Verbreitung gefunden hat und in vielen Bereichen der Softwareentwicklung einen großen Stellenwert besitzt (vgl. [Barc90, HaKoHo95, Sche05]). Auch die Sprache BASIC, die erste vom Autor erlernte Programmiersprache, unterstützt die prozedurale Programmierung (vgl. [Boon83]).

Bei der prozeduralen Programmierung wird ein serielles, imperatives Programm in mehrere wiederverwendbare Unterprogramme aufgeteilt. Diese Unterprogramme können für die jeweilige Ausführung Parameter entgegennehmen und werden Prozeduren genannt. Ein prozedurales Programm nutzt den Arbeitsspeicher in Form von Variablen in einem globalen Kontext. In Prozeduren können auch lokale Variablen verwendet werden, deren Speicher nach der Ausführung der Prozedur wieder freigegeben wird.

Prozeduren sind ein Hilfsmittel, um imperative Programme zu strukturieren und den Quellcode wiederverwendbar zu machen. Wird eine Gruppe von Prozeduren und Variablen zu einer Einheit zusammengefasst, nennt man das *Modul*.

2.1.4 Objektorientierte Programmierung

Das Paradigma der Objektorientierung beschreibt bis auf einige Ausnahmen nicht den primären Charakter einer Sprache, in der ein Problem oder seine Lösungsstrategie beschrieben wird, sondern es gibt ein Rahmenwerk für die Strukturierung der Problemlösung vor.

So ist z. B. das funktionale Programmierparadigma ein Konzept, das beschreibt, wie eine Lösungsstrategie beschrieben wird – nämlich durch das Formulieren von Funktionen. Ebenso ermöglicht das logische Programmierparadigma die Formulierung eines Programms als Menge logischer Aussagen. Das Constraint-basierte Paradigma hingegen verwendet Regelwerke. Nach dem Paradigma der objektorientierten Programmierung wird ein Programm als Menge kommunizierender Objekte beschrieben.

Die objektorientierte Programmierung wird nur in wenigen Sprachen (z. B. SmallTalk, vgl. [Mitt97]) als reines bzw. primäres Paradigma umgesetzt. Die meisten objektorientierten Sprachen kombinieren es mit einem weiteren Programmierparadigma (vgl. *Tabelle 2.1*). Aktuell ist

ein Trend hin zu Multiparadigmen-Sprachen mit mehr als zwei primären Programmierparadigmen zu beobachten, manifestiert durch die Sprachen Scala und F#, welche das funktionale, das imperative und das objektorientierte Paradigma unterstützen (vgl. [SyGrCi07, WaPa09]).

Wenn von objektorientierten Programmiersprachen gesprochen wird, sind meist die imperativen, objektorientierten Programmiersprachen gemeint.

Das folgende Zitat von Grady Booch fasst den Kern der objektorientierten Programmierung gut zusammen:

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. [Booc94, S. 38]

Einen kurzen Abriss der Objektorientierung hat der Autor bereits in [Kier08, S. 13ff] formuliert. Im Folgenden soll jedoch etwas detaillierter auf das Thema eingegangen werden.

Kapselung und Klassen

Neben dem Aspekt, dass die Objektorientierung das Programm selbst strukturiert, gibt es die Beziehungen zwischen Programm und Daten vor. Die Objektorientierung sieht vor, dass Daten und Programmteile in der Art zu Einheiten zusammengefasst werden, dass zu jeder Gruppe von Daten direkt jene Programmteile gehören, welche diese Daten verarbeiten. Eine strikte Umsetzung der Objektorientierung bedeutet weiterhin, dass nur jene Programmteile, welche zu den Daten gehören, auch Zugriff auf die Daten erhalten. Dadurch entsteht das Prinzip der *Kapselung*.

Eine solche Einheit aus Daten und verantwortlichen Progammteilen wird als *Objekt* bezeichnet. In der Regel werden die Datenstruktur und die Programmteile von Objekten in sog. *Klassen* beschrieben. Es können beliebig viele Objekte einer Klasse erzeugt werden. Ein Objekt ist eine *Instanz* einer Klasse.

Zustand und Verhalten von Objekten

Die Werte, die von der Datenstruktur eines Objekts gespeichert werden, repräsentieren den Zustand eines Objektes (vgl. [Booc91, S. 78-80]). Objekte können Nachrichten austauschen. Beim Empfang einer Nachricht, kann ein Objekt wiederum anderen Objekten Nachrichten senden und den eigenen Zustand verändern. Die Reaktion eines Objektes auf eine Nachricht nennt man *Verhalten* (vgl. [Booc91, S. 80-83]).

Der Nachrichtenaustausch erfolgt überlicherweise über Methoden. Ruft ein Objekt A die Methode eines Objektes B auf, wird dadurch eine Nachricht übermittelt und das Objekt B erhält

die Möglichkeit mit seinem Verhalten auf die Nachricht zu reagieren. Durch die Definition von Methoden für eine Klasse wird vorgeschrieben, welche Nachrichten die Objekte der Klasse empfangen können.

Vererbung zwischen Klassen

Da Objekte sich häufig ähneln oder zumindest eine Anzahl gemeinsamer Eigenschaften und Verhaltensmuster besitzen, wurde das Konzept der *Vererbung* von Klassen entwickelt (vgl. [Booc91, S. 97-114]). Dabei kann eine Klasse die Datenstrukturen und Verhaltensmuster einer anderen Klasse erben und diese erweitern oder durch eigenes Verhalten ersetzen.

Am Beispiel von Früchten haben Birnen und Äpfel viele Gemeinsamkeiten, aber jedes Kind kann einen Apfel von einer Birne unterscheiden. Die Gemeinsamkeiten können z. B. in der Klasse Obst zusammengefasst werden. Die Klasse der Äpfel und die Klasse der Birnen erbt das Verhalten der Klasse Obst und erweitert oder variiert es. Die Klasse Obst wiederum erbt von der Klasse Frucht.

Abstraktion durch Vererbung

Aus dem Prinzip der Vererbung folgt das Konzept der *Abstraktion*. Dieses Konzept besagt, dass von einem Objekt nicht die konkrete Klasse bekannt sein muss, um Nachrichten an das Objekt zu senden. Es muss nur *eine* Klasse aus der Vererbungshierarchie der konkreten Klasse des Objektes bekannt sein, um alle in dieser Klasse definierten Methoden für das konkrete Objekt aufrufen zu können.

In dem Beispiel von Apfel und Birne bedeutet das, dass von einer Frucht nicht bekannt sein muss, ob sie ein Apfel ist, um sie essen zu können. Es reicht aus zu wissen, dass sie Obst ist.

Sichtbarkeiten, Schnittstellen und Geheimnisprinzip

Bei Klassen, die ein kompliziertes Verhalten implementieren, ist es sinnvoll das Verhalten derart zu strukturieren, dass es innerhalb der Klasse wiederverwendet werden kann und dass der Programmierer das Verhalten leichter versteht. Dazu werden Verhaltensmuster in einzelne Methoden aufgespalten. Diese sollen aber i. d. R. nicht von außen erreichbar sein. Für derartige Szenarien wurden *Sichtbarkeiten* in der Objektorientierung eingeführt. Sichtbarkeiten ermöglichen dem Programmierer für jede Methode festzulegen, ob sie nur innerhalb der Klasse aufgerufen werden darf bzw. sichtbar ist, oder ob andere Objekte von außen Zugriff haben.

Alle von außen sichtbaren Methoden einer Klasse ergeben die *Schnittstelle* der Klasse. In vielen objektorientierten Programmiersprachen ist es möglich eine Schnittstelle ohne eine

konkrete Implementierung zu beschreiben. Soll eine Klasse eine Schnittstelle implementieren, muss sie die notwendigen Methoden öffentlich zur Verfügung stellen.

Ist von einem konkreten Objekt bekannt, dass seine Klasse eine Schnittstelle implementiert, können die Methoden aufgerufen werden, ohne dass bekannt sein muss, wie sie in der konkreten Klasse implementiert sind. Diese Eigenschaft der Objektorientierung wird *Geheimnisprinzip* genannt.

Durch das Geheimnisprinzip wird es möglich, die Abhängigkeiten zwischen größeren Teilen einer Software mit Schnittstellen zu beschreiben und die konkrete Implementierung bei Bedarf auszutauschen. Durch das Geheimnisprinzip können Programmteile leichter wiederverwendet werden.

2.1.5 Imperative, objektorientierte Programmierung

Bei der imperativen, objektorientierten Programmierung, verkörpert durch Programmiersprachen wie C++, Java, C# (vgl. *Tabelle 2.1*), werden die Möglichkeiten der Strukturierung des Programms durch die Objektorientierung genutzt. Diese Sprachen ermöglichen aber ebenso die imperative Programmierung.

Öffentliche Felder

Für die oben genannten Sprachen gilt, dass – wie in der objektorientierten Programmierung üblich – Klassen und Methoden definiert werden können. Darüber hinaus können jedoch Felder öffentlich deklariert werden, sodass ein direktes Ändern des Zustands eines Objektes möglich ist, ohne eine Methode aufzurufen. Diese Möglichkeit wirkt dem Prinzip der Kapselung (vgl. *Kapselung und Klassen* auf Seite 17) entgegen.

Statische Klassenmitglieder

Zusätzlich ist es in den oben genannten Sprachen möglich, sog. statische Felder und Methoden zu definieren. Der Speicherbereich für statische Felder wird nicht für jedes Objekt bei der Instanziierung bereitgestellt, sondern ist Teil der Klasse. Dadurch ist der Wert des Feldes unabhängig von den Instanzen der Klasse und wird gleichsam zwischen allen Instanzen geteilt. Statische Felder können auch öffentlich definiert werden, damit sie den globalen Variablen der prozeduralen Programmierung ähneln.

Auch Methoden können statisch definiert werden, sodass es keiner Instanz bedarf, um sie aufzurufen. Statische Methoden verkörpern nicht das Verhalten eines Objektes beim Empfangen einer Nachricht, sondern gleichen vielmehr den Prozeduren der prozeduralen Programmierung.

Mit Hilfe von statischen Elementen kann auch in einer Sprache wie Java oder C# nahezu vollständig ohne Objektorientierung und damit prozedural programmiert werden (vgl. *Unterabschnitt 2.1.3 Prozedurale Programmierung*).

2.2 Programmiersprachen

Um einige Beispiele für die in *Abschnitt 2.1 Programmierparadigmen* genannten Programmierparadigmen zu geben, werden in diesem Abschnitt einige Programmiersprachen genannt. Dabei werden diejenigen Programmiersprachen als Beispiele ausgewählt, mit denen der Autor bereits kleinere oder größere Projekte umgesetzt hat.

In *Tabelle 2.1* werden die Programmiersprachen einander gegenüber gestellt. Dabei werden für jede Sprache die unterstützten Programmierparadigmen angegeben (vgl. *Abschnitt 2.1 Programmierparadigmen*).

Bei einigen Sprachen ist diese Klassifizierung nicht ganz einfach. Denn in vielen Sprachen ist es möglich, eine Abstraktionsschicht mit Hilfe der von der Programmiersprache zur Verfügung gestellten Mittel einzurichten. Durch eine solche Abstraktionsschicht kann die Programmierung in einem Programmierparadigma erfolgen, welches von der Sprache selbst nicht unterstützt wird. Diese Möglichkeit ist nicht nur rein theoretischer Natur, sondern wird durchaus ernsthaft diskutiert (vgl. [AbGu04, Wiki10a]). In der folgenden Tabelle werden deshalb nur jene Programmierparadigmen ausgewiesen, welche von der Sprache aktiv – z. B. durch Schlüsselwörter – unterstützt werden.

Tabelle 2.1: Programmiersprachen

Name	imperativ	objektorientiert	funktional	logisch
C	●			
C++	●	●		
C#	●	●		
F#	●	●	●	
Java	●	●		
Prolog				●
Scala	●	●	●	
Scheme			●	
SmallTalk		●		
VB.NET	●	●		

2.3 Microsoft .NET-Framework

In diesem Abschnitt soll das Microsoft .NET-Framework kurz vorgestellt und ein Abriss über seine Entwicklung gegeben werden. Im praktischen Teil dieser Arbeit sollen die Klassenbi-

bliotheken des .NET-Frameworks in das datenflussorientierte Programmiersystem DynamicNodes eingebunden werden. Die Technologie des .NET-Frameworks ist nicht nur für die einzubindenen Klassenbibliotheken von Bedeutung, es ist zusätzlich die technische Basis für das datenflussorientierte Programmiersystem DynamicNodes. Somit spielt das .NET-Framework eine wichtige Rolle im praktischen Teil dieser Arbeit. Ausführlichere Darstellungen finden sich in [Kier08, S. 16ff] und [ScHuBr06]. Im Anschluss werden einige Technologien aus dem Umfeld des .NET-Frameworks vorgestellt, die für diese Arbeit besondere Bedeutung besitzen.

2.3.1 Entwicklung des .NET-Frameworks

Der Software-Hersteller Microsoft hat schon in einem frühen Stadium seines Betriebssystems Windows ein Framework für Software-Komponenten mit dem Namen Component Object Model (COM) etabliert. Dieses Framework wurde für die Entwicklung von Komponenten mit der Programmiersprache C++ konzipiert. Die meisten Programmierschnittstellen von Windows sind auch heute in der aktuellen Version 7 als COM-Schnittstellen ausgelegt.

Später wurde bei Microsoft versucht, dieses Framework für verteilte Anwendungen zu erweitern. Das Ergebnis dieser Bemühungen war das Distributed Component Object Model (DCOM). Auch eine Weiterentwicklung des ursprünglichen COM: das COM+, wurde den Programmierern als neue Technik für die komponentenorientierte Programmierung vorgestellt. Jedoch wuchs mit der fortschreitenden Weiterentwicklung auch die Komplexität der Technologie und zusätzlich erkannte Microsoft, dass es die Programmierer für die Windows-Plattform nicht ausschließlich an die Programmiersprache C++ fesseln konnte.

Die Folge war die Entwicklung eines neuen Komponenten-Frameworks, welches das in die Jahre gekommene COM+ ablösen sollte. Ein wesentliches Entwicklungsziel der neuen Plattform war die Möglichkeit, mit verschiedenen Sprachen Komponenten implementieren zu können, die dennoch reibungslos miteinander interagieren. Microsoft beobachtete die Entwicklung der Sprache Java und versuchte neben einer Weiterentwicklung von C++, unter dem Namen C#, und einer Weiterentwicklung von VisualBasic, unter dem Namen VB.NET, auch einen Dialekt von Java, damals in der Version 1.1 unter dem Namen J#, als Sprache für das neue Framework zu etablieren. Jedoch konnte Microsoft die Sprache J# nicht in dem Tempo weiterentwickeln, wie es bei dem ursprünglichen Java der Fall war. In Folge dessen ließen sich nur wenige Java-Programmierer von der neuen Plattform begeistern.

Obwohl das neue Framework als Nachfolger der COM-Technologie gedacht war, waren die Unterschiede zum Vorgänger doch so groß, dass Microsoft sich für einen eigenen Namen entschied. Der Begriff „.net” (lies dott-nett) mit dem dazugehörigen Logo (vgl. *Abbildung 2.4*) war geboren.

Abbildung 2.4: Microsoft .NET Logo (l. bis 2008 und r. aktuell)

Um die Komposition von Komponenten zu ermöglichen, die in verschiedenen Programmiersprachen implementiert wurden, war ein gemeinsames und stabiles Typensystem sehr wichtig. Dieses Typensystem – von Microsoft Common Type System (CTS) genannt – war die Grundlage für die öffentlichen Schnittstellen der Komponenten. Der wichtigste Schritt, der das .NET-Framework von der COM-Technologie abhob, war die Einführung einer virtuellen Maschine, der Common Language Runtime (CLR). Die CLR ermöglichte die Abstraktion der Implementierung von dem Betriebssystem und der Hardware, sodass es prinzipiell möglich wurde, die kompilierten Programme auf unterschiedlichen Betriebssystemen und Hardwareplattformen auszuführen. Microsoft produzierte von der CLR jedoch lediglich eine Version, die unter Windows lief. Eine Implementierung der Laufzeitumgebung für andere Betriebssysteme wurde erst später durch das OpenSource-Projekt Mono (vgl. [Nove10]) in Angriff genommen.

Die Abstraktion von der Hardware mit Hilfe der CLR war nur möglich durch die Einführung einer maschinennahen Sprache, die von der Laufzeitumgebung während der Ausführung ohne großen Mehraufwand in echte Maschinenbefehle übersetzt werden konnte. Diese Sprache nannte Microsoft Intermediate Language (IL). Alle .NET-kompatiblen Compiler, z. B. für C#, VB.NET oder J# hatten die Aufgabe IL-Code zu produzieren.

Durch den Versuch die Sprache Java in Form von J# mit einzubeziehen, hatte die erste Version von .NET noch große Ähnlichkeiten mit der damaligen Java-Plattform. Eine Gemeinsamkeit ist z. B. der Einsatz eines Garbage Collector (GC) für die automatische Speicherverwaltung.

Im Laufe der folgenden Jahre entwickelten sich die beiden Technologien jedoch zunehmend in verschiedene Richtungen. Während Java die Sprache für viele Betriebssysteme wurde und den Schwerpunkt auf das Internet in Form von Applets und Serveranwendungen legte, wurde das .NET-Framework die Plattform für viele Sprachen und fasste neben den serverseitigen Internetanwendungen mehr in der Client-Programmierung Fuß. Diese grobe Richtung verblasst aber bei den aktuellen Versionen von Java (Version 6 – vgl. [Orac10]) und .NET (Version 4 – vgl. [Micr10a]), denn beide Technologien hatten ausreichend Zeit in der Breite zu wachsen.

Bisher sind die folgenden Versionen des .NET-Frameworks erschienen: 1.0 (Februar 2002), 1.1 (April 2003), 2.0 (Januar 2006), 3.0 (November 2006), 3.5 (November 2007), 3.5 SP1 (August 2008), 4.0 (April 2010). Eine Untersuchung zum Wachstum der .NET-Klassenbibliothek ist in *Abschnitt A.2 Benchmark für Bindungstechniken* im Anhang auf Seite 145 zu finden.

2.3.2 Besonderheiten der .NET-Klassen

Die Klassen der .NET-Klassenbibliotheken basieren auf dem CTS des .NET-Frameworks und weisen zwei Besonderheiten auf, die in *Unterabschnitt 2.1.2 Imperative Programmierung*, *Unterabschnitt 2.1.4 Objektorientierte Programmierung* und *Unterabschnitt 2.1.5 Imperative, objektorientierte Programmierung* nicht behandelt wurden. Dazu gehören sog. Eigenschaften (*Properties*) und Ereignisse (*Events*).

Eigenschaften

Die Eigenschaften von .NET-Klassen vereinen die z. B. aus Java bekannten Get- und Set-Methoden (*Getter/Setter*) in einem einzigen Klassenmitglied. Ihre Aufgabe ist es, einen benannten Wert von einem Objekt für andere Objekte verfügbar zu machen, ohne ein Feld zu veröffentlichen. Der Programmierer einer Eigenschaft kann für das Setzen und das Lesen der Eigenschaft jeweils einen Code-Block formulieren. Die Eigenschaft kann mit Hilfe dieser Code-Blöcke ein privates Feld nach außen verfügbar machen. Die Code-Blöcke können den Wert, der gelesen und geschrieben wird, aber auch auf andere Weise speichern, z. B. in der Eigenschaft eines weiteren Objektes (*Proxy*) oder direkt in einer Datei.

In C# gibt es seit der Sprachversion 3.0 zwei Schreibweisen für eine Eigenschaft. Die Kurzschreibweise (vgl. *Listing 2.1*, Z. 3) wird im folgenden Code-Beispiel durch die Eigenschaft `Stunden` demonstriert. Sie setzt sich zusammen aus einem Zugriffsmodifizierer (`public`), einem Datentyp (`int`) und einem Namen (`Stunden`). Der Rumpf der Eigenschaft besteht nur aus `{ get; set; }`, was soviel bedeutet wie: „Diese Eigenschaft kann gelesen und geschrieben werden."

Die ausführliche Schreibweise, demonstriert durch die Eigenschaft `Minuten`, zeigt das technische Wesen einer Eigenschaft deutlicher. Diese Eigenschaft speichert ihren Wert in dem sog. *backing field* `m_minuten`. Auch in der ausführlichen Schreibweise (vgl. *Listing 2.1*, Z. 5 - 9) besteht die Definition einer Eigenschaft aus Zugriffsmodifizierer, Datentyp und Name. Im Rumpf werden die Schlüsselwörter `get` und `set` jedoch jeweils mit einem eigenen Code-Block ausgestattet. Der Code-Block von `get` kann beliebigen Code enthalten, muss aber am Ende mit `return` einen Wert vom Datentyp der Eigenschaft zurückgeben. Der Code-Block von `set` kann ebenfalls beliebigen Code enthalten. Der Code kann den neuen Wert für die Eigenschaft über das Schlüsselwort `value` abrufen.

Listing 2.1: .NET-Eigenschaft in C# ausführlich und in Kurzform

```
1 class Uhr
2 {
3     public int Stunden { get; set; }
4
```

```
5       private int m_minuten;
6       public int Minuten
7       {
8           get { return m_minuten; }
9           set{ m_minuten = value;}
10      }
11  }
```

Der C#-Compiler baut die Eigenschaften nicht nur in ihrer eigentlichen Form zum Lesen und Schreiben in die Klasse ein. Interessant ist, dass zusätzlich im Hintergrund, für den C#-Programmierer unsichtbar, Methoden-Signaturen für die Code-Blöcke von Getter und Setter erzeugt werden. Die Methode für den Code-Block des Getters folgt dabei dem Muster `<Datentyp> get_<Name>()` und die Methode für den Code-Block des Setters folgt dem Muster `void set_<Name>(<Datentyp> newValue)`.

Die Existenz von Getter und Setter, in Form von echten Methoden, für eine .NET-Eigenschaft lässt sich leicht in der PowerShell nachweisen (vgl. *Listing 2.2*). Zunächst wird über die statische Eigenschaft `Now` der Klasse `System.DateTime` eine Instanz von `System.DateTime` mit dem aktuellen Zeitpunkt erzeugt. Anschließend wird mit dem Befehl `Get-Member` eine Liste aller Mitglieder von `System.DateTime` ausgegeben. In dieser Liste (hier verkürzt dargestellt) findet sich eine Eigenschaft `Hour` vom Datentyp `System.Int32` aber keine Methode mit der Signatur `System.Int32 get_Hour()`. Wird versucht diese Methode aufzurufen, offenbart sich jedoch ihre verdeckte Existenz.

Listing 2.2: PowerShell-Sitzung – Existenz der Get-Methode für Eigenschaften

```
1  PS> $time = [System.DateTime]::Now
2  PS> Get-Member -InputObject $time
3
4     TypeName: System.DateTime
5
6  Name                 MemberType     Definition
7  ----                 ----------     ----------
8  ...
9  Hour                 Property       System.Int32 Hour {get;}
10 ...
11
12 PS> Write-Host $time.Hour
13 11
14 PS> Write-Host $time.get_Hour()
15 11
```

Die automatisch vom C#-Compiler erzeugten Getter- und Setter-Methoden für die .NET-Eigenschaften ermöglichen es, die Eigenschaften auf Software-technischer Ebene als Metho-

denpaare zu betrachten. Eigenschaften können auch statisch definiert werden.

Ereignisse

Die zweite Besonderheit von .NET-Klassen sind Ereignisse. Ereignisse setzen das Observer-Design-Pattern (vgl. [GaHeJoVl94]) direkt im Typensystem des .NET-Frameworks um. Dazu kann eine Klasse ein Ereignis definieren. Die Signatur eines Ereignisses setzt sich zusammen aus einem Zugriffspezifizierer (`public`), dem Schlüsselwort `event`, dem Delegaten-Typ (Ein Typ, der die Signatur der Methoden beschreibt, die bei dem Ereignis registriert werden dürfen – `EventHandler`.) und einem Namen (`Kräht`).

```
class Hahn
{
        public event EventHandler Kräht;
        ...
}
```

Eine Methode kann bei einem Ereignis registriert werden, wenn ihre Signatur dem Delegaten-Typ des Ereignisses entspricht. Wird das Ereignis ausgelöst, wird die registrierte Methode ausgeführt.

```
class Bauer
{
        private Hahn m_hahn;
        public Bauer()
        {
                m_hahn = new Hahn();
                m_hahn.Kräht += Aufwachen;
        }
        private void Aufwachen(object sender, EventArgs e)
        {
                Console.WriteLine("Aufgewacht!");
        }
}
```

Um ein Ereignis auszulösen, muss das Objekt, welches das Ereignis besitzt, zunächst überprüfen, ob das Ereignis durch eine Methode überwacht wird. Wenn dies der Fall ist, kann es ausgelöst werden.

```
class Hahn
{
        public event EventHandler Kräht;

        public void MeldeSonnenaufgang()
```

```
        {
                if (Kräht != null)
                {
                        Kräht.Invoke(this, EventArgs.Empty);
                }
        }
}
```

Soll die Überwachung eines Ereignisses abgebrochen werden, muss die registrierte Methode wieder abgemeldet werden.

```
class Bauer
{
        ...
        private void Aufwachen(object sender, EventArgs e)
        {
                Console.WriteLine("Aufgewacht!");
                m_hahn.Kräht -= Aufwachen;
        }
}
```

Das Mittel in C# für das Registrieren bei einem Ereignis ist der Operator `+=` und für das Abmelden der Operator `-=`. Ähnlich wie bei den Eigenschaften werden vom C#-Compiler aber für jedes Ereignis im Hintergrund zwei versteckte Methoden erzeugt. Eine Methode für das Anmelden mit der Signatur `void add_<Name>(<Delegaten-Typ> handler)`. Und eine Methode für das Abmelden mit der Signatur `void remove_<Name>(<Delegaten-Typ> handler)`. Auch die Existenz dieser Methoden lässt sich in der PowerShell nachweisen, wie *Listing 2.3* zeigt.

Zunächst wird die Klassenbibliothek für die Programmierung eines Graphical User Interface (GUI) (`System.Windows.Forms`) geladen. Anschließend wird ein Fenster erzeugt und ein Titelzeilentext zugewiesen. Der entscheidende Punkt ist die Zeile 4. Hier wird die vom Compiler für das Ereignis `Click` erzeugte Methode `void add_Click(EventHandler handler)` aufgerufen. Zu guter Letzt wird eine Nachrichtenbehandlungsschleife für das Fenster gestartet. Zur Demonstration wurde zweimal in das Fenster geklickt. Die Ausgabe in der PowerShell (Zeile 6 u. 7) weist nach, dass der Handler für das Ereignis tatsächlich registriert wurde.

Listing 2.3: PowerShell-Sitzung – Existenz der Add-Methode für Ereignisse

```
1 PS> $null = [System.Reflection.Assembly]::LoadWithPartialName("System.
      Windows.Forms")
2 PS> $form = New-Object System.Windows.Forms.Form
3 PS> $form.Text = "Ereignistest"
4 PS> $form.add_Click( [EventHandler] { Write-Host "Click!" } )
```

```
PS> [System.Windows.Forms.Application]::Run($form)
Click!
Click!
```

In .NET ist es auch möglich statische Ereignisse in Klassen zu definieren.

2.3.3 Für diese Arbeit wichtige .NET-Technologien

Einige Technologien des .NET-Frameworks haben für diese Arbeit eine besondere Bedeutung. Diese sollen hier kurz vorgestellt werden.

Reflection-API

Das .NET-Framework stellt ein Application Programming Interface (API) zur Verfügung, mit dem es möglich ist, Metadaten von kompilierten Klassenbibliotheken und Programmen auszuwerten. Auch das Laden und Ausführen von zur Entwicklungszeit unbekannten Klassenbibliotheken ist damit möglich. Die Fähigkeit eines Programms, seinen eigenen Programmcode analysieren zu können, wird Reflektion genannt. Aus diesem Grund heißt das API im .NET-Framework für diesen Zweck *Reflection-API*.

Die Reflection-API des .NET-Frameworks stellt im Namensraum `System.Reflection`[1] Klassen für die Verarbeitung der Metadaten von Klassenbibliotheken zur Verfügung. Ausgehend von der Klasse `System.Type`[2] können über die Methoden `GetConstructors()`, `GetEvents()`, `GetFields()`, `GetMethods()` und `GetProperties()` `MemberInfo`[3]-Objekte abgerufen werden, welche die Mitglieder des Typs näher beschreiben.

Für jede Art von Klassenmitglied existiert eine von `MemberInfo` abgeleitete Klasse mit spezifischen Fähigkeiten. So gibt die Methode `GetMethods()` z. B. Objekte der Klasse `MethodInfo`[4] zurück, welche von der Klasse `MemberInfo` erben. Jedes `MethodInfo`-Objekt ermöglicht das Abfragen von Eigenschaften der beschriebenen Methode. So bietet die Klasse `MethodInfo` z. B. Eigenschaften wie `string Name`, `bool IsPrivate`, `bool IsPublic`, `bool IsStatic`, `Type DeclaringType`, `Type ReturnType` und die Methode `ParameterInfo[] GetParameters()`.

Über die Eigenschaften der Klassenmitglieder hinaus bieten die `MemberInfo`-Objekte aber auch die Möglichkeit mit den Klassenmitgliedern zu interagieren. So besitzt die Klasse `MethodInfo` z. B. die überladene Methode `Object Invoke(Object obj, Object[] parameters)`[5], welche einen indirekten Aufruf einer durch das `MethodInfo`-Objekt beschriebenen

[1] http://msdn.microsoft.com/en-us/library/136wx94f.aspx
[2] http://msdn.microsoft.com/en-us/library/system.type.aspx
[3] http://msdn.microsoft.com/en-us/library/system.reflection.memberinfo.aspx
[4] http://msdn.microsoft.com/en-us/library/system.reflection.methodinfo.aspx
[5] http://msdn.microsoft.com/en-us/library/a89hcwhh.aspx

Methode erlaubt. Analog besitzt die Klasse `PropertyInfo` die zwei Methoden `Object GetValue(Object obj, Object[] index)`[1] und `void SetValue(Object obj, Object value, Object[] index)`[2]. Mit diesen Methoden kann der Wert einer statischen oder dynamischen Eigenschaft gelesen und geschrieben werden.

IL-Emitter / Lambda-Ausdrücke / C#-Compiler

Das .NET-Framework bietet drei verschiedene Möglichkeiten neuen IL-Code zur Laufzeit in das laufende Programm einzubringen und auszuführen.

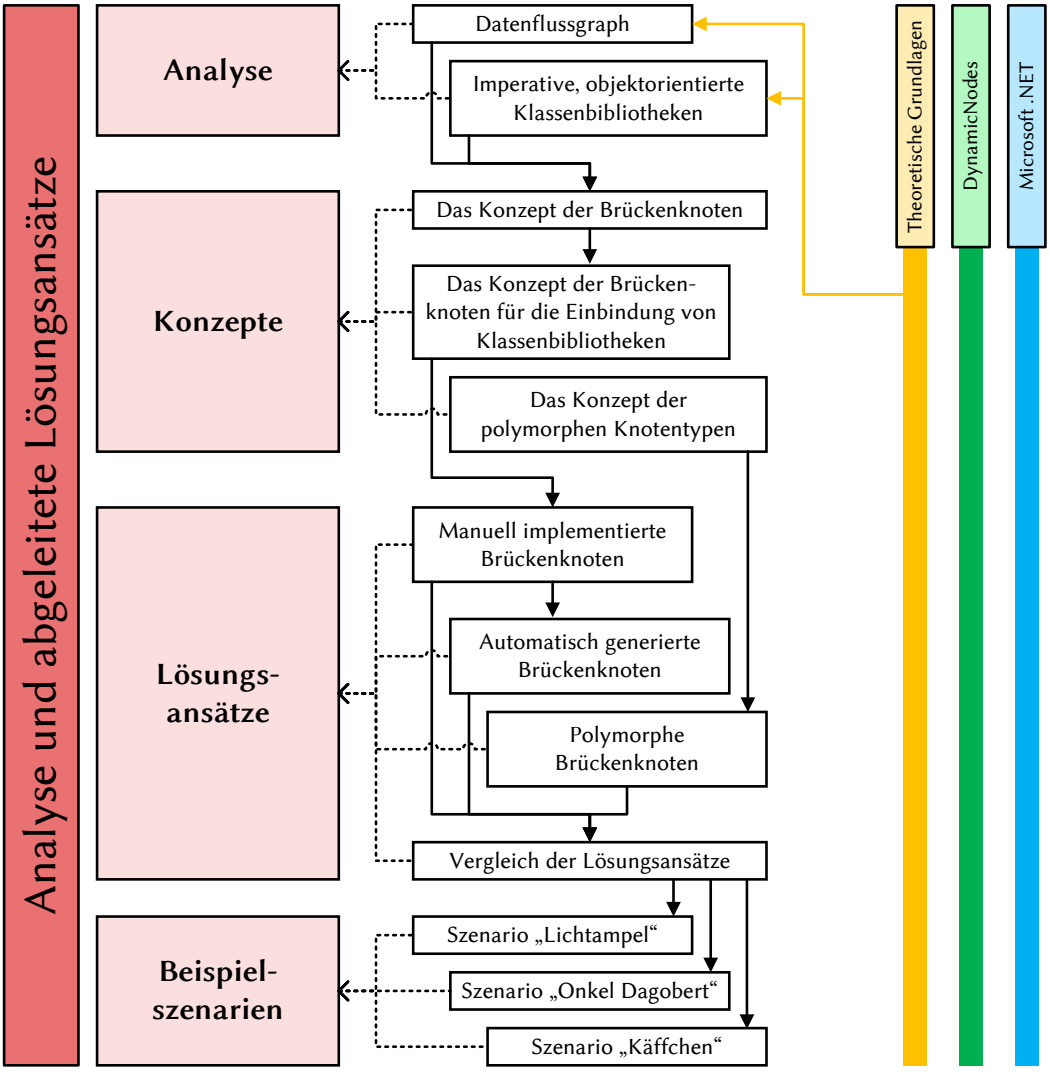
Die erste Möglichkeit ist Teil der Reflection-API und wird Code-Emittierung genannt. Die beiden wichtigsten Klassen für diesen Prozess sind die Klassen `System.Reflection.Emit.DynamicMethod`[3] und `System.Reflection.Emit.ILGenerator`[4]. Mit diesen beiden Klassen kann die Signatur einer Methode durch Name, Parameter und Rückgabewert und der Methodenrumpf durch eine Sequenz von IL-Befehlen zur Laufzeit definiert werden. Dabei wird keine Hochsprache wie C# oder VisualBasic.NET verwendet, sondern der IL-Code wird direkt in Form der erforderlichen Metadatenstrukturen (`AssemblyBuilder`, `TypeBuilder`, `MethodBuilder`, `ParameterBuilder`)[5] und den sog. Op-Codes (`System.Reflection.Emit.OpCodes`[6]) der IL definiert.

Ab der Version 3.5 des .NET-Frameworks steht eine weitere Möglichkeit zum Erzeugen von Methoden zur Laufzeit zur Verfügung. Mit den statischen Methoden der Klasse `Expression`[7] und der Klasse `Expression<TDelegate>`[8] im Namensraum `System.Linq.Expressions` ist es möglich, einen Lambda-Ausdruck als Baumstruktur zur Laufzeit aufzubauen und mit der Methode `TDelegate Compile()`[9] der Klasse `Expression<TDelegate>` in IL-Code zu kompilieren.

Da der C#-Compiler in das .NET-Framework integriert ist, ist es auch möglich zur Laufzeit eine Zeichenkette zu konstruieren, welche einen C#-Quelltext enthält. Dieser Quelltext kann mit Hilfe der Klasse `Microsoft.CSharp.CSharpCodeProvider`[10], welche von `System.CodeDom.Compiler.CodeDomProvider`[11] erbt, in IL-Code übersetzt werden.

[1] http://msdn.microsoft.com/en-us/library/b05d59ty.aspx
[2] http://msdn.microsoft.com/en-us/library/xb5dd1f1.aspx
[3] http://msdn.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod.aspx
[4] http://msdn.microsoft.com/en-us/library/system.reflection.emit.ilgenerator.aspx
[5] http://msdn.microsoft.com/en-us/library/xd5fw18y.aspx
[6] http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes_fields.aspx
[7] http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx
[8] http://msdn.microsoft.com/en-us/library/bb335710.aspx
[9] http://msdn.microsoft.com/en-us/library/bb345362.aspx
[10] http://msdn.microsoft.com/en-us/library/microsoft.csharp.csharpcodeprovider.aspx
[11] http://msdn.microsoft.com/en-us/library/system.codedom.compiler.codedomprovider.aspx

3 Analyse und abgeleitete Lösungsansätze



Dieses Kapitel gliedert sich in vier Abschnitte. Zunächst werden in Abschnitt 3.1 die beiden Welten der datenflussorientierten Programmierung und der imperativen, objektorientierten Programmierung analysiert. Dabei wird der Schwerpunkt auf die Ein- und Ausgabefähigkeiten von Programmen in dem jeweiligen Programmierparadigma gelegt. Denn für den Entwurf einer Brücke zwischen den beiden Programmierparadigmen müssen die Ausgaben eines Datenflussgraphen als Eingaben für ein imperatives, objektorientiertes Programm verwendet werden und umgekehrt. Ein Programm in der datenflussorientierten Programmierung ist ein Datenflussgraph und ein Programm in der imperativen, objektorientierten Programmierung wird durch eine Klassenbibliothek verkörpert. Neben dem Datenaustausch wird die Kontrolle des Programmablaufs in den beiden Programmierparadigmen untersucht.

In Abschnitt 3.2 werden zwei Konzepte vorgestellt. Das Konzept der Brückenknoten und das Konzept der polymorphen Knotentypen. Das Konzept der Brückenknoten wird zusätzlich zu einer allgemeinen Darstellung in einer spezialisierten Form für die Einbindung von imperativen, objektorientierten Klassenbibliotheken beschrieben. Diese Spezialisierung ist die Grundlage für den praktischen Teil der Arbeit.

Aufbauend auf den Konzepten, die im zweiten Abschnitt dieses Kapitels beschrieben wurden, werden in Abschnitt 3.3 drei Lösungsansätze für die Realisierung einer Brücke zwischen einem datenflussorientierten Programmiersystem und imperativen, objektorientierten Klassenbibliotheken vorgestellt und verglichen.

Um die verschiedenen Lösungsansätze aus dem dritten Abschnitt im Umfeld eines konkreten Szenarios zu bewerten und damit eine Hilfestellung für den Entwurf einer Brücke zu liefern, werden in Abschnitt 3.4 drei konkrete Szenarien beschrieben, ein passender Lösungsansatz ausgewählt und die Auswahl begründet.

3.1 Analyse

In diesem Abschnitt werden zunächst die Begriffe Interaktion und Interaktionsform definiert. Anschließend werden der Datenflussgraph und die imperative, objektorientierte Klassenbibliothek genauer untersucht.

Der Begriff *Interaktion* zwischen zwei Systemen bedeutet hier: Datenaustausch zwischen den Systemen und/oder Ausführung von Unterprogrammen in einem System, ausgelöst durch das andere System.

Eine Klasse von Interaktionen, bei der alle Interaktionen gleicher Natur sind, sich die einzelnen Interaktionen aber auf einen unterschiedlichen Kontext beziehen, wird *Interaktionsform* genannt.

3.1.1 Datenflussgraph

Die Struktur eines Datenflussgraphen besteht aus Knoten und Verbindungen zwischen den Knoten. Diese Struktur bietet keine Interaktionsmöglichkeiten, um Daten hinein oder heraus zu transportieren (vgl. *Unterabschnitt 2.1.1 Datenflussorientierte Programmierung*). Um dennoch Daten mit einem Datenflussgraphen auszutauschen, werden Knoten verwendet, deren Operationen diese Arbeit übernehmen.

Soll ein Datenstrom zur Verarbeitung in einen Graphen hineingeleitet werden, z. B. von einer Netzwerkschnittstelle oder aus einer Datei, ist ein Knoten (Quelle) erforderlich, der die Daten von der Netzwerkschnittstelle oder aus der Datei liest und sie über einen Ausgang innerhalb des Graphen weitergibt. Auch für die Ausgabe von Daten aus einem Datenflussgraphen ist ein Knoten (Senke) erforderlich, der die auszugebenden Daten über einen Eingang innerhalb des Graphen entgegennimmt und z. B. in eine Datei schreibt oder auf dem Bildschirm ausgibt.

Es lässt sich die Schlussfolgerung ziehen, dass die Kommunikation zwischen einem Datenflussgraphen und der Außenwelt oder zwischen einem Datenflussgraphen und einem Fremdsystem über spezialisierte Knoten realisiert werden muss. Die Interaktionsformen eines Datenflussgraphen ergeben sich dabei aus denjenigen Knoten des Graphen, welche Daten aus dem Graphen heraus und in den Graphen hinein transportieren können.

Gelingt es einen Knoten zu implementieren, der mit einem fremden System interagiert, ist eine Brücke zu diesem System geschaffen.

3.1.2 Imperative, objektorientierte Klassenbibliothek

In diesem Abschnitt werden imperative, objektorientierte Klassenbibliotheken untersucht. Zunächst wird analysiert, wie eine Interaktion ausgeführt werden kann (Bindung). Anschließend werden die verschiedenen Interaktionsformen betrachtet.

Eine imperative, objektorientierte Klassenbibliothek bietet viele verschiedene Interaktionsformen. Diese lassen sich in drei Gruppen aufteilen, wobei jede Gruppe einem Programmierparadigma zugeordnet ist. Diese Gruppen sind objektorientierte, imperative und prozedurale Interaktionen.

Bindung

Damit eine imperative, objektorientierte Klassenbibliothek von einem System verwendet werden kann, benötigt es eine Form der Bindung. Eine Bindung besteht dann, wenn zu dem Zeitpunkt, zu dem die Interaktion ausgeführt werden soll, alle notwendigen Informationen und Algorithmen in einer für das System ausführbaren Form vorliegen.

Damit eine Interaktion in einer imperativen, objektorientierten Klassenbibliothek von einem System ausgeführt werden kann, muss die Interaktion benannt bzw. in irgendeiner Form identifiziert sein. Diese Identität wird durch eine *Adresse* beschrieben. In einer Klassenbibliothek gehört eine Interaktion zu einer Klasse. Die Adresse einer Interaktion ist i. d. R. relativ zu der Klasse. Demzufolge wird auch für die Klasse eine Adresse benötigt, um die Interaktion eindeutig zu identifizieren.

Eine Adresse für eine Klasse oder eine Interaktion kann aus einem Namen oder einer relativen Speicher- bzw. Einsprungadresse bestehen. Ein Name in Form einer Zeichenkette ist dann möglich, wenn die Klassenbibliothek Informationen für die Abbildung der Namen auf die numerischen Adressen mitbringt (*Metadaten*).

Ist die Adresse einer Interaktion bekannt, müssen alle Eingabedaten für die Interaktion in einer Form aufbereitet werden, in der sie von der Interaktion verarbeitet werden können. Soll z. B. ein Java-Programm eine .NET-Methode aufrufen und dabei Parameter übergeben, müssen die Parameterwerte, welche in Form von Java-Datentypen vorliegen, in .NET-Datentypen umgewandelt werden.

Umgekehrt müssen Rückgabewerte von Interaktionen aus der Technologie der Klassenbibliothek in die Technologie des interagierenden Systems umgewandelt werden. Im Sinne des eben genannten Beispiels müssen also Rückgabewerte von .NET-Datentypen in Java-Datentypen umgewandelt werden.

Unter bestimmten Umständen ist die Bindung besonders einfach zu realisieren. Nämlich genau dann, wenn das System, welches mit der imperativen, objektorientierten Klassenbibliothek interagieren soll, mit der Klassenbibliothek technologisch verwandt ist, d. h. mit der gleichen Technologie implementiert ist wie die Klassenbibliothek.

Soll z. B. ein Java-Programm an eine Java-Bibliothek angebunden werden, beschränkt sich die Bindung auf einen direkten Aufruf im Quellcode. Soll hingegen ein Java-Programm an eine .NET-Klassenbibliothek angebunden werden – das Java-Programm ist der .NET-Klassenbibliotheken technologiefremd – wird eine technologische Brücke benötigt.

Das Umformen von Daten zwischen technologiefremden Systemen wird *Marshalling* genannt.

Objektorientierter Anteil

Der objektorientierte Anteil der Interaktionen mit einer Klassenbibliothek beschränkt sich auf das Instanzieren von neuen Objekten für eine Klasse aus der Bibliothek und den Aufruf von Methoden auf diesen Objekten (vgl. *Zustand und Verhalten von Objekten* auf Seite 17). In manchen Sprachen, wie z. B. SmallTalk, sind Klassen selbst wiederum Objekte. In einem

solchen Fall ist auch die Instanzierung lediglich ein Methodenaufruf auf einem Objekt. Die objektorientierten Interaktionsformen einer Klassenbibliothek bestehen also mindestens aus dem Methodenaufruf auf einem Objekt und in vielen Sprachen zusätzlich aus der Instanzierung von neuen Objekten einer Klasse.

Für einen Methodenaufruf ist eine Referenz auf das Objekt erforderlich, auf dem die Methode aufgerufen werden soll. Zusätzlich ist der Name bzw. die Adresse der Methode erforderlich und ggf. eine Reihe von Werten für die Parameter der Methode.

Für eine Instanzierung gilt im Prinzip das Gleiche, mit dem Unterschied, dass bei Sprachen mit Klassendefinitionen, die keine Objekte sind, nicht eine Objektreferenz, sondern die Adresse der Klassendefinition erforderlich ist.

Imperativer Anteil

Der imperative Anteil der Interaktionsformen mit Klassenbibliotheken prägt sich unter anderem durch die Möglichkeit aus, Felder eines Objektes zu publizieren, sodass andere Objekte diese von außen lesen oder schreiben können, ohne eine Methode aufzurufen (vgl. *Öffentliche Felder* auf Seite 19). Dadurch können Objekte ihren gegenseitigen Zustand verändern, ohne sich mittels Methoden darüber zu informieren und sich somit eine Reaktionsmöglichkeit zu geben (vgl. *Zustand und Verhalten von Objekten* auf Seite 17). Der imperative Anteil der Interaktionsformen mit Klassenbibliotheken besteht also zum Teil aus dem Lesen und Schreiben von öffentlichen Objektfeldern.

Um ein Objektfeld lesen zu können, wird lediglich die Objektreferenz und der Name bzw. die Adresse des Feldes benötigt. Um ein Objektfeld schreiben zu können, ist zusätzlich ein neuer Wert erforderlich.

Darüber hinaus spielt das Ausführungsmodell von imperativen Programmen eine wesentliche Rolle. Das imperative Programmierparadigma arbeitet mit einem sequentiellen Programmablauf. Das bedeutet, dass für die Nutzung einer imperativen Programmierschnittstelle, wie sie imperative Klassenbibliotheken bilden, eine definierte zeitliche Reihenfolge der Interaktionen erforderlich ist.

Prozeduraler Anteil

Die prozedurale Programmierung wird von Klassenbibliotheken dann unterstützt, wenn sich Variablen und Methoden verwenden lassen, ohne eine Instanz einer Klasse erzeugen oder benutzten zu müssen. Diese Variablen und Methoden ohne Objektkontext werden als statische Klassenmitglieder bezeichnet (vgl. *Statische Klassenmitglieder* auf Seite 19). Eine statische Methode entspricht einer Prozedur.

Für die statischen Elemente einer Klassenbibliothek gilt das Gleiche wie für Methoden und Objektfelder, mit dem Unterschied, dass es für die Interaktion keiner Objektreferenz als Kontext bedarf.

Für den Aufruf einer Prozedur ist also lediglich der Name bzw. die Einsprungadresse und eine Anzahl von Werten für die ggf. benötigten Parameter erforderlich. Für das Lesen und Schreiben von statischen Feldern wird nur der Name bzw. die numerische Adresse und beim Schreiben ein neuer Wert benötigt.

3.2 Konzepte

Bevor auf konkrete Lösungsansätze eingegangen wird, sollen allgemeine Konzepte vorgestellt werden, welche die Grundlage für die verschiedenen Lösungsansätze bilden. Dazu gehören die Brückenknoten – als allgemeines Konzept und konkretisiert zur Anbindung einer imperativen, objektorientierten Klassenbibliothek – und die polymorphen Knotentypen.

3.2.1 Das Konzept der Brückenknoten

Wie in *Unterabschnitt 3.1.1 Datenflussgraph* erläutert, besteht die einzige Möglichkeit für einen Datenflussgraphen mit der Außenwelt zu interagieren darin, Knoten zu verwenden, die diese Aufgabe übernehmen. Ein Knoten, der die Interaktion mit einem Fremdsystem übernimmt, wird Brückenknoten genannt (vgl. *Abbildung 3.1*).

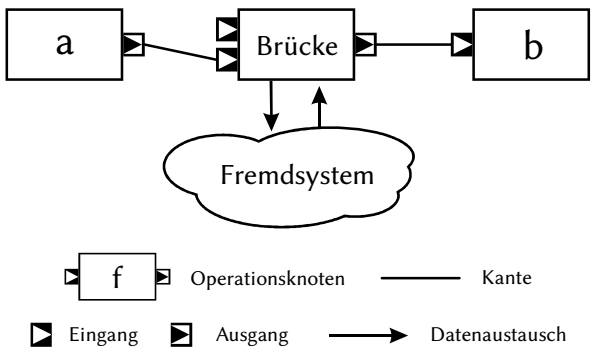


Abbildung 3.1: Ein Brückenknoten, der Daten mit einem Fremdsystem austauscht

Zur Veranschaulichung kann das folgende Beispiel dienen: Ein Datenflussgraph soll Daten in ein relationales Datenbanksystem schreiben. Dazu wird ein Knoten implementiert, welcher über einen Eingang Daten-Tupel entgegennimmt, mit diesen einen SQL-Befehl bildet und an ein Datenbanksystem sendet. Dieser Knoten ist ein Brückenknoten zu einer relationalen Datenbank (vgl. *Abbildung 3.2*).

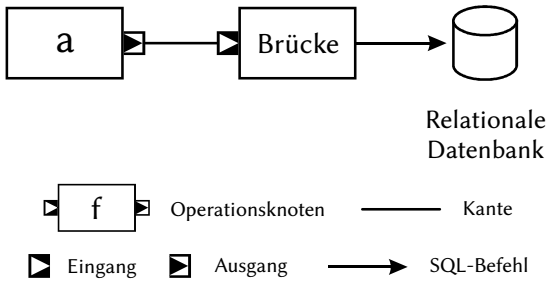


Abbildung 3.2: Ein Brückenknoten zu einer relationalen Datenbank

3.2.2 Das Konzept der Brückenknoten für die Einbindung von imperativen Klassenbibliotheken

Brückenknoten zwischen einem Datenflusssystem und einer imperativen, objektorientierten Klassenbibliothek müssen, abgeleitet von den in *Unterabschnitt 3.1.1 Datenflussgraph* und *Unterabschnitt 3.1.2 Imperative, objektorientierte Klassenbibliothek* erläuterten Interaktionsformen, die im Folgenden beschriebenen Anforderungen erfüllen.

Ausgehend von der Definition am Anfang von *Kapitel 3 Analyse und abgeleitete Lösungsansätze* kann der Begriff *Interaktion*, im Kontext des Konzeptes von Brückenknoten, zu einer imperativen Klassenbibliothek konkretisiert werden: Mit dem Begriff Interaktion werden Instanzierung, statische und objektgebundene Methodenaufrufe und das Lesen und Schreiben von statischen Feldern und Objektfeldern zusammengefasst.

Anbindung an einen Datenflussgraphen

Jede Interaktion zwischen einem Datenflussgraphen und einer imperativen Klassenbibliothek muss durch einen Knoten in dem Datenflussgraphen realisiert werden (vgl. *Unterabschnitt 3.1.1 Datenflussgraph*).

Instanzierung von Objekten

Die Brücke muss die Möglichkeit bieten, neue Objekte zu erzeugen. Für diese Operation muss die Adresse der Klasse bzw. die Referenz auf das Klassenobjekt ausgewählt werden können und es müssen ggf. Werte für die Parameter des Konstruktors über Eingänge eines Knotens entgegengenommen werden. Die Referenz auf das erzeugte Objekt muss über einen Ausgang weitergegeben werden.

Aufruf von Methoden

Die Brücke muss die Möglichkeit bieten, eine Methode auf einem Objekt aufzurufen. Für diese Operation muss die Adresse einer Klasse oder die Referenz auf das Klassenobjekt und die Adresse einer Methode ausgewählt werden können. Es müssen eine Objektreferenz und ggf. Werte für Parameter über Eingänge eines Knotens entgegengenommen werden.

Sind Parameter Referenzparameter, d. h. erhält eine Methode bei Übergabe eines Parameterwertes eine Refererenz auf den Speicherbereich, in dem der Parameterwert gespeichert ist, und hat damit die Möglichkeit den Parameterwert durch ein Beschreiben des referenzierten Speichers gleichsam für den Aufrufer zu verändern, muss der veränderte Wert über einen Knotenausgang weitergegeben werden.

Auch die Ergebnisse einer Methode in Form von Rückgabewerten oder Ausgabeparametern müssen über Knotenausgänge weitergegeben werden.

Lesen von Feldern

Die Brücke muss die Möglichkeit bieten, den Wert eines Objektfeldes auszulesen. Dazu muss die Adresse einer Klasse oder die Referenz auf das Klassenobjekt und die Adresse des Feldes ausgewählt werden können. Des Weiteren muss über einen Knoteneingang eine Objektreferenz entgegengenommen werden. Der jeweils aktuelle Wert des gelesenen Feldes muss über einen Knotenausgang weitergegeben werden.

Schreiben von Feldern

Die Brücke muss die Möglichkeit bieten, den Wert eines Objektfeldes zu verändern. Dazu muss die Adresse einer Klasse oder die Referenz auf das Klassenobjekt und die Adresse des Feldes ausgewählt werden können. Des Weiteren müssen über Knoteneingänge eine Objektreferenz und der neue Wert für das Feld entgegengenommen werden.

Interagieren mit statischen Mitgliedern

Die Brücke muss die Möglichkeit bieten, eine statische Methode aufzurufen. Abgesehen von der nicht erforderlichen Objektreferenz gilt das Gleiche wie für den nicht statischen Aufruf von Methoden.

Die Brücke muss die Möglichkeit bieten, ein statisches Feld zu lesen. Abgesehen von der nicht erforderlichen Objektreferenz gilt das Gleiche wie für das Lesen von Objektfeldern.

Die Brücke muss die Möglichkeit bieten, ein statischen Feld zu schreiben. Abgesehen von der nicht erforderlichen Objektreferenz gilt das Gleiche wie für das Schreiben von Objektfeldern.

Einbettung in einen Kontrollfluss

Die Brücke muss die Möglichkeit bieten, die Reihenfolge von Interaktionen zu definieren. Falls das Datenflussprogrammiersystem die Definition eines Kontrollflusses mit Hilfe von Kanten und Steuermarken erlaubt, muss die Brücke entsprechende Ein- und Ausgänge anbieten, um auf den Kontrollfluss reagieren zu können und die Steuermarken auf geeignete Weise weiterzuleiten.

3.2.3 Das Konzept der polymorphen Knotentypen

Ein *polymorpher Knotentyp* ist ein Knotentyp, dessen Knoten durch Parametrisierung, während oder nach der Instanzierung, unterschiedliche Ein- und Ausgänge zur Verfügung stellen und in der Lage sind, die eingehenden Daten entsprechend der Parametrisierung zu verarbeiten. Ein polymorpher Knotentyp ist z. B. dann sinnvoll, wenn eine Klasse mit unbekannter Anzahl von Operationen existiert, die in ein Datenflusssystem eingebunden werden sollen.

Ein polymorpher Knotentyp arbeitet typischerweise in zwei Phasen: In der Initialisierungsphase werden die Ein- und Ausgänge eingerichtet bzw. die Konfiguration von existierenden Ein- und Ausgängen angepasst. Diese Phase kann sich in den parametrisierten Instanzierungsvorgang des Knotens einbetten oder auch später über eine Konfigurationsschnittstelle des Knotens ausgelöst werden. Im zweiten Fall wird nicht nur von einem *polymorphen Knotentyp* sondern auch von *polymorphen Knoten* gesprochen. Die Initialisierungsphase kann bei polymorphen Knoten auch mehrmahls durchgeführt werden, sodass sich die Ein- und Ausgänge des Knotens während der Laufzeit verändern.

Die zweite Phase ist die Ausführungsphase. Wird der Knoten aktiviert, muss zunächst durch eine Fallunterscheidung, die der aktuellen Parametrisierung und den vorhandenen Ein- und Ausgängen entsprechende Operation ausgewählt werden. Erst danach kann diese ausgeführt werden.

Zur Veranschaulichung kann das folgende Beispiel dienen: Ein Knoten soll einzelne Werte zu einem Tupel vereinen. Wie viele Werte in dem Tupel zusammengefasst werden sollen, ist zur Zeit der Implementierung des Knotens noch nicht bekannt. Ein polymorpher Knoten, der diese Aufgabe erfüllt, nimmt als Parameter während der Instanzierung eine natürliche Zahl entgegen. In der Initialisierungsphase erzeugt er so viele Eingänge wie der Parameter vorgibt und einen Ausgang für den Ergebnistupel. Während der Operation erzeugt der Knoten einen Tupel, dessen Datentyp von der Anzahl der Eingänge abhängig ist, füllt die Werte von den Eingängen in die einzelnen Elemente des Tupels und gibt diesen über den Ausgang weiter.

In Datenflusssystemen, in denen für jeden Anschluss eine Kompatibilität definiert wird, lassen sich zwei Arten von Polymorphie unterscheiden. Ein Knotentyp ist *schwach polymorph*,

wenn während seiner Initialisierungsphase ausschließlich die Kompatibilität von Anschlüssen angepasst wird. Ein Knotentyp ist *voll polymorph*, wenn während seiner Initialisierungsphase Anschlüsse hinzugefügt oder entfernt werden.

Ob ein Datenflusssystem *polymorphe Knotentypen* unterstützt, hängt von den folgenden Voraussetzungen ab:

- Die Instanzierung eines Knotens aus einem Knotentyp kann parametrisiert werden.
- Die Ein- und Ausgänge können, durch einen Algorithmus des Knotentyps, für jede Knoteninstanz unterschiedlich eingerichtet werden.

Ob ein Datenflusssystem auch *polymorphe Knoten* unterstützt, hängt von den folgenden Voraussetzungen ab:

- Die Knoten verfügen über eine Schnittstelle, mit der sie nach der Instanzierung konfiguriert werden können, ohne Daten in einen Eingang zu schicken.
- Die Knoten können nach der Instanzierung Anschlüsse ändern, bzw. erzeugen und entfernen.

Ein Datenflusssystem lässt sich mit dem folgenden Schema nach Polymorphie auf Knotenebene klassifizieren:

- Kann das System die Kompatibilität von Anschlüssen zur Laufzeit ändern?
    - nein – Das System unterstützt *keine Polymorphie auf Knotenebene.*
    - ja – Kann das System Anschlüsse zur Laufzeit erzeugen und entfernen?
        - nein – Kann das System die Initialisierungsphase nach der Instanzierung durchführen?
            - nein – Das System unterstützt *schwach polymorphe Knotentypen.*
            - ja – Das System unterstützt *schwach polymorphe Knoten.*
        - ja – Kann das System die Initialisierungsphase nach der Instanzierung durchführen?
            - nein – Das System unterstützt *voll polymorphe Knotentypen.*
            - ja – Das System unterstützt *voll polymorphe Knoten.*

Ein Beispiel für die Definition eines polymorphen Knotens in Pseudo-Code findet sich in *Listing 3.1.*

Listing 3.1: Polymorpher Knoten als Pseudo-Code

```
1 Knotendefinition
2 {
```

```
3         Anzahl: Zahl
4         Eingänge: Liste
5         Ausgänge: Liste
6
7         Initialisierung(tupelGröße: Zahl)
8         {
9                 Anzahl := tupelGröße
10                Von i := 0 bis tupelGröße - 1
11                {
12                        Eingänge.FügeHinzu(neu Eingang())
13                }
14                Ausgänge.FügeHinzu(neu Ausgang())
15        }
16
17        Arbeit()
18        {
19                tupel: Tupel
20                tupel := new Tupel(Anzahl)
21                Von i := 0 bis Anzahl - 1
22                {
23                        tupel[i] := Eingänge[i].NeuerWert
24                }
25                Ausgänge[0].Sende(tupel)
26        }
27 }
```

3.3 Lösungsansätze

Aufbauend auf den Konzepten in *Abschnitt 3.2 Konzepte* werden in diesem Abschnitt konkrete Lösungsansätze vorgestellt, die für die Einbindung einer imperativen, objektorientierten Klassenbibliothek in ein Datenflussprogrammiersystem geeignet sind. Alle Lösungsansätze stützen sich auf das Konzept aus *Unterabschnitt 3.2.2 Das Konzept der Brückenknoten für die Einbindung von imperativen Klassenbibliotheken.*

3.3.1 Manuell implementierte Brückenknoten

Eine Möglichkeit eine Anzahl von Brückenknoten zu realisieren, ist die manuelle Implementierung. Dabei muss der Programmierer für jede Interaktion mit einer imperativen, objektorientierten Klassenbibliothek, die er in einem Datenflussprogrammiersystem verwenden möchte, einen eigenen Brückenknoten implementieren.

Am Beispiel eines Methodenaufrufs umfasst eine Implementierung die folgenden Aspekte:

- Die „harte" Kodierung der Adresse der Klasse
- Die Definition eines Eingangs für eine Objektreferenz
- Die „harte" Kodierung der Adresse der Methode innerhalb der Klasse
- Die Definition von jeweils einem Eingang für jeden Methodenparameter
- Die Definition von jeweils einem Ausgang für jeden Rückgabewert, Ausgabe oder Referenzparameter
- Die Operation, welche die adressierte Methode auf dem erhaltenen Objekt aufruft, dabei die Parameter mit Werten von den Eingängen belegt und den Rückgabewert und Werte von Ausgabe- und Referenzparameter über die Ausgänge weitergibt.

Das bedeutet, dass die Anzahl der zu implementierenden Knoten genauso groß ist, wie die benötigen Interaktionen. Sollen von einem Datenflusssystem z. B. 20 Klassen mit jeweils 4 Methoden und 2 Feldern benutzt werden, sind je Klasse – angenommen für jede Klasse ist eine Instanzierung notwendig – 9 Interaktionen zu implementieren (je Feld sind zwei Interaktionen erforderlich – Lesen und Schreiben). Das führt in diesem Beispiel zu einer Anzahl von 180 spezialisierten Knoten, die alle manuell implementiert werden müssen. Bei der Implementierung von mehreren Interaktionen der gleichen Form – z. B. Brückenknoten für 10 verschiedene Methoden – ist die Ähnlichkeit der einzelnen Implementierungen sehr groß. Folglich entsteht viel redundanter Code, der manuell gepflegt werden muss.

Nicht nur der Aufwand der Implementierung von einer großen Anzahl von Brückenknoten kann zu Problemen führen, auch die Verwaltung der dabei entstehenden Menge von Knotentypen in dem Datenflusssystem kann problematisch sein. Dabei stellt die Benutzerschnittstelle ein größeres Problem dar, als die technische Verwaltung (Speicher, Rechenleistung). Denn eine große Anzahl von Knotentypen kann dazu führen, dass die Benutzeroberfläche unübersichtlich und kompliziert wird.

Ist für die Erzeugung von lauffähigen Knotentypen eine Kompilierung der Knotentypdefinitionen notwendig, so ist für eine Erweiterung der Anbindung durch neue Interaktionen stets ein Kompiliervorgang und damit in vielen Fällen eine Unterbrechung der Ausführung des Datenflusssystems notwendig.

Ein Vorteil dieses Ansatzes ist, dass die Komplexität der Implementierung erwartungsgemäß recht gering ist. Weiterhin vorteilhaft ist, dass die Implementierung evtl. für einzelne Interaktionen optimiert werden kann. Soll z. B. ein Brückenknotentyp für eine Methode mit vielen Parametern implementiert werden und ist vorher bereits bekannt, dass für einen erheblichen Teil der Parameter im Kontext der Anbindung an das Datenflusssystem immer die gleichen Werte verwendet werden, brauchen diese Parameter nicht als Eingänge publiziert werden, sondern können durch eine „harte" Kodierung der Standardwerte verdeckt bleiben.

Ob die anzubindende Klassenbibliothek maschinenlesbare Metainformationen über ihre Klassen und Klassenmitglieder mitbringt, ist für diesen Lösungsansatz unerheblich, da die Adressierung von Klassen und Klassenmitgliedern durch den Programmierer vorgenommen wird und dieser in der Lage ist, nur menschlich lesbare Metainformationen für die Implementierung zu verwenden.

Pro

- Niedriger Aufwand für wenige Interaktionen
- Optimale Anpassung der Knotentypen an die Interaktionen
- Anbindung von Klassenbibliotheken ohne maschinenlesbare Metainformationen ist möglich
- Funktioniert mit Datenflusssystemen, die keine polymorphen Knoten unterstützen

Kontra

- Sehr großer Aufwand für eine größere Anzahl von Interaktionen
- Datenflusssystem muss eine große Anzahl Kontentypen verwalten
- Viel redundanter Code
- Evtl. Kompilierung notwendig
- Statische Bindung zwischen Datenflusssystem und Klassenbibliothek

3.3.2 Automatisch generierte Brückenknoten

Eine Möglichkeit, die Nachteile des manuellen Ansatzes aus *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* zu verringern, ist ein Programm (Code-Generator), welches den Quelltext für die Definition der Brückenknoten automatisch generiert. Das Beispiel für die durch eine Implementierung abzudeckenden Aspekte aus *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* lässt sich ohne Anpassung übernehmen.

Die automatische Generierung des Quelltextes für die Definitionen der Brückenknoten hat zur Folge, dass der Aufwand für die Implementierung von der Anzahl der anzubindenden Interaktionen entkoppelt wird. Der Aufwand für die Implementierung des Code-Generators ist jedoch um ein vielfaches höher als der für die Implementierung eines einzelnen Brückenknotens. Aus diesem Grund eignet sich dieser Ansatz gut für Fälle, in denen die Anzahl der anzubindenden Interaktionen groß, aber bekannt ist.

Damit der Code-Generator die Definitionen für die Brückenknoten automatisch produzieren kann, müssen Metainformationen, z. B. die Adressen von Klassen und Klassenmitgliedern, in maschinenlesbarer Form vorliegen. Der vom Code-Generator erzeugte Quelltext zur Definition der Brückenknoten kann nachträglich manuell angepasst werden. Deshalb ist die Flexibilität dieses Ansatzes gut.

Ob die anzubindende Klassenbibliothek technologisch mit dem Datenflusssystem verwandt ist, spielt bei automatisch generiertem Quellcode eine nicht ganz so große Rolle wie bei dem manuellen Ansatz, da der Aufwand von der Anzahl der Brückenknoten unabhängig ist. Die Anbindung einer Klassenbibliothek ist auch mit automatisch generierten Brückenknoten statisch. Das bedeutet, dass bei Erweiterung der Menge der anzubindenen Interaktionen evtl. eine erneute Kompilierung notwendig wird.

Die Menge der erzeugten Knotentypen unterscheidet sich im Vergleich zum Ansatz aus *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* aber nicht, sodass es immer noch zu Problemen bei der Verwaltung der Knotentypen in dem Datenflussprogrammiersystem kommen kann.

Pro

- Aufwand der Implementierung von der Anzahl der Interaktionen unabhängig
- Funktioniert mit Datenflusssystemen, die keine polymorphen Knoten unterstützen
- Anpassung der Knotentypen an die Interaktionen möglich

Kontra

- Hoher Implementierungsaufwand für die Code-Generierung
- Kompilieren für neue Interaktionen notwendig

3.3.3 Polymorphe Brückenknoten

Unterstützt ein Datenflusssystem polymorphe Knotentypen, eröffnet sich eine weitere Möglichkeit: Jede Interaktionsform (Instanzierung, Methodenaufruf, Felder Lesen und Schreiben, etc.) (vgl. *Unterabschnitt 3.1.2 Imperative, objektorientierte Klassenbibliothek*) kann mit nur einem polymorphen Brückenknoten abgedeckt werden (vgl. *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen*). Dabei wird der Variabilität mehrerer Interaktionen einer Interaktionsform – z. B. die Parameteranzahl von verschiedenen Methoden – nicht manuell wie in *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* und auch nicht durch einen

Code-Generator wie in *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten*, sondern durch einen Algorithmus in der Initialisierungsphase des polymorphen Knotentyps begegnet.

Am Beispiel eines Methodenaufrufs wird dem polymorphen Knotentyp bei der Instanzierung die Adresse einer Klasse und die Adresse der Methode in der Klasse übergeben. In der Initialisierungsphase legt der Knoten für die notwendige Objektreferenz einen Eingang `contextRef` und für jeden Parameter von `i=1..n` einen Eingang `inParam<i>` an. Zusätzlich legt er für den Rückgabewert der Methode einen Ausgang `result` und für jeden Referenz- bzw. Ausgabeparameter von `j=1..m` einen Ausgang `outParam<j>` an.

Während der Ausführungsphase benutzt der polymorphe Knoten den Wert des Eingangs `contextRef` als Kontext für den Methodenaufruf und übergibt die Werte der Eingänge `inParam<i>` für `i=1..n` als Parameter an die Methode. Nach Abschluss des Methodenaufrufs übergibt er das Ergebnis an den Ausgang `result` und die Werte der Referenz- und Ausgabeparameter an die Ausgänge `outParam<j>` für `j=1..m`.

Die Implementierung eines polymorphen Brückenknotens muss die folgenden Aspekte abdecken:

- Parametrisierung des Knotentyps mit der Adresse der Klasse
- Parametrisierung des Knotentyps mit der Adresse des Klassenmitglieds
- Dynamische Erzeugung der Ein- und Ausgänge für Objektkontext, Parameter und Rückgabewerte
- Durchführung der Interaktion mit Hilfe der Adressen und Eingabewerte
- Weitergabe der Ergebniswerte an die Ausgänge

Für die Implementierung der Initialisierungsphase müssen neben der Adresse einer Klasse und des anzubindenden Klassenmitglieds Metadaten zu den Eigenschaften des adressierten Mitglieds verfügbar sein. Z. B. muss der Algorithmus der Initialisierung die Anzahl der Parameter einer Methode ermitteln und entscheiden, ob die Methode einen Rückgabewert liefert oder nicht. Arbeitet die Klassenbibliothek mit einem statischen Typensystem, müssen auch die Datentypen der Parameter und des Rückgabewertes ermittelt werden, um die Ein- und Ausgänge des Knotens entsprechend zu konfigurieren und die Übergabe von falsch typisierten Daten zu verhindern.

Ein Nachteil dieses Ansatzes ist der Overhead, der während der Ausführungsphase auftritt. Da der Knotentyp für den Brückenknoten nicht speziell für jede Interaktion kompiliert wird, muss die Bindung und Ausführung der Interaktion dynamisch erfolgen und die Werte der Eingänge müssen in Abhängigkeit der Metainformationen den Parametern zugeordnet werden.

Die dynamische Bindung ist nur möglich, wenn eine der folgenden Bedingungen erfüllt ist:

1. Die anzubindende Klassenbibliothek ist in einer interpretierten Sprache implementiert, sodass auch die Interaktionen von dem polymorphen Brückenknoten als Zeichenfolge generiert und von der Laufzeitumgebung der Skriptsprache interpretiert werden können. Diese Bedingung erfüllt z. B. die Sprache JavaScript.
2. Die anzubindende Klassenbibliothek ist in einer dynamisch typisierten Sprache implementiert und die Laufzeitumgebung der Sprache bietet eine Schnittstelle, um die Interaktionen durchzuführen. Diese Bedingung erfüllt z. B. die Sprache Python.
3. Die anzubindende Klassenbibliothek basiert auf einem statischen Typensystem und es existiert eine Programmierschnittstelle, mit der die Metadaten der Klassenbibliothek gelesen, ausgewertet und die Interaktionen durchgeführt werden können. Eine solche Programmierschnittstelle wird auch *Reflection-API* genannt. Diese Bedingung wird z. B. von der Java-Laufzeitumgebung und von dem Microsoft .NET-Framework erfüllt.

Da bei diesem Ansatz die Anzahl der Knotentypen der Anzahl der Interaktionsformen entspricht und damit für einen Menschen leicht überschaubar bleibt, ist die Verwaltung der Knotentypen in dem Datenflusssystem und die Benutzeroberfläche nicht mit den in *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* beschriebenen Problemen behaftet.

Pro

- Geringer Aufwand für eine große Anzahl von Interaktionen
- Kein Kompilieren für neue Interaktionen notwendig
- Überschaubare Anzahl von Knotentypen

Kontra

- Relativ komplexe Knoten
- Begrenzte Anpassungsfähigkeiten für einzelne Interaktionen
- Overhead

Als besondere Form der polymorphen Brückenknoten lässt sich ein Brückenknoten betrachten, der durch seine Polymorphie nicht nur verschiedene Interaktionen einer Interaktionsform anbinden kann, sondern auch alle Interaktionsformen unterstützt. Solch ein universeller polymorpher Brückenknoten bringt den Vorteil mit sich, dass er als einzelner Knotentyp eine vollständige Brücke zu einer imperativen, objektorientierten Klassenbibliothek bildet. Der Nachteil ist die gesteigerte Komplexität der Implementierung im Vergleich zu polymorphen Brückenknoten je Interaktionsform.

Tabelle 3.1: Eigenschaften der Lösungsansätze

	manuell (technologisch verwandt)	automatisch	polymorph (universell)
Implementierungsaufwand für wenige Interaktionen	⟹ (⇓)	⇑	⇗ (⇑)
Implementierungsaufwand für viele Interaktionen	⇑ (⇑)	⇘	⇓ (⇘)
Aufwand hängt von der Anzahl der Interaktionen ab	●	●	○
Anzahl der Knotentypen	⇑ (⇑)	⇑	⇘ (⇓)
Kompilierung für weitere Interaktionen notwendig	●	●	○
Anpassungsfähigkeit an einzelne Interaktionen	⇑ (⇑)	⇗	⇘ (⇓)
Unterstützung für polymorphe Knotentypen notwendig	○	○	●
Maschinenlesbare Metainformationen notwendig	○	●	●
Overhead zur Laufzeit	⟹ (⇓)	⟹	⇗ (⇑)

Legende
⇑, ⇗, ⟹, ⇘, ⇓ – Tendenzen von „sehr hoch" bis „sehr tief"
● – Aussage trifft zu
○ – Aussage trifft nicht zu

3.3.4 Vergleich der Lösungsansätze

Für den Vergleich der Lösungsansätze aus *Abschnitt 3.3 Lösungsansätze* sollen zunächst die Eigenschaften in *Tabelle 3.1* zusammengetragen werden. Da bei dem Vergleich der Ansätze für einige Eigenschaften keine absoluten Aussagen, z. B. über Implementierungsaufwand oder Anzahl der Knotentypen, gemacht werden können, werden diese jeweils mit Tendenzen bewertet. Ist eine Eigenschaft eine boolsche Aussage, wird eine genaue Aussage getroffen.

In *Tabelle 3.1* ist gut zu erkennen, dass keiner der Ansätze grundsätzlich ausscheidet oder den übrigen Ansätzen grundsätzlich überlegen ist. Jeder Ansatz hat seine Vorzüge und seine Nachteile. Aus diesem Grund lässt sich einem der Ansätze nur für eine konkrete Konstellation von Rahmenbedingungen der Vorzug geben.

Grundsätzlich gegensätzlich verhalten sich die manuellen Ansätze gegenüber den Ansätzen mit automatischer Code-Generierung oder polymorphen Brückenknoten. In *Abbildung 3.3* wird in Form eines Diagramms gezeigt, wie sich der Implementierungsaufwand der Ansätze über die Anzahl der einzubindenden Interaktionen entwickelt. Die Achsen des Diagramms

sind mit keiner Skala versehen, weil die Aussagen nur eine Tendenz darstellen und keine absoluten Werte angegeben werden können.

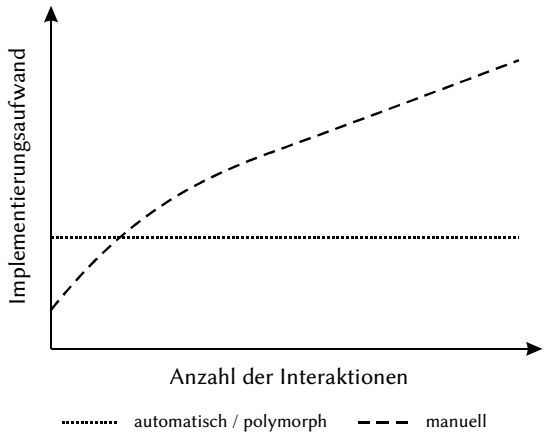


Abbildung 3.3: Implementierungsaufwand verschiedener Lösungsansätze

3.4 Beispielszenarien

Um eine Entscheidungsgrundlage für eine Implementierung zu bieten, sollen hier drei Beispielszenarien beschrieben werden und in dem Kontext des jeweiligen Szenarios eine Bewertung der Ansätze und die Auswahl eines geeigneten Ansatzes erfolgen.

3.4.1 Szenario 1 – „Lichtampel"

Das erste Szenario ist typisch für die Einbindung einer kleinen Programmierschnittstelle in ein Datenflussprogrammiersystem. Es könnte sich um die API einer USB-Steuerung für eine Steckdosenleiste handeln.

Beschreibung des Szenarios

Von einer Klassenbibliothek sollen zwei Klassen mit jeweils vier Methoden eingebunden werden. Die Technologie der Klassenbibliothek ist mit der Technologie des Datenflusssystems nicht verwandt. Das Datenflusssystem unterstützt polymorphe Knotentypen.

Auswahl eines Lösungsansatzes

Alle Lösungsansätze kommen theoretisch in Frage. Es sollen lediglich 10 Interaktionen eingebunden werden. Deshalb ist die Zeile „Implementierungsaufwand für wenige Interaktionen"

eine der bedeutendsten für die Entscheidung, welcher Ansatz am besten geeignet ist. Mit einer mittleren Tendenz erzeugt der manuelle Ansatz (vgl. *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten*) den geringsten Aufwand und sollte deshalb näher untersucht werden.

Dass bei dem manuellen Ansatz der Implementierungsaufwand von der Anzahl der Interaktionen abhängt, ist bei 10 Interaktionen weniger ein Nachteil denn ein Vorteil. Die Anzahl der Knotentypen entspricht der Anzahl der Interaktionen – auch das ist bei 10 Interaktionen kein Problem. Ein Vorteil der manuellen Methode ist, dass die Knotentypen genau an die jeweiligen Bedürfnisse der Interaktionen angepasst werden können. Obwohl polymorphe Knotentypen möglich wären, kann deren Komplexität und ihr Overhead in diesem Fall vermieden werden.

Dass bei Erweiterung der Anbindung auf weitere Interaktionen neu kompiliert werden muss, ist in diesem Szenario nicht relevant, da sich die Anzahl der einzubindenden Interaktionen nicht häufig ändert. Die Performance der Brückenknoten ist relativ gut, da eine statische Bindung mit den Interaktionen durchgeführt wird. Allerdings existiert ein Technologiebruch bei der Bindung der Interaktion und dessen Overhead ist hier unbekannt.

Für die in *Unterabschnitt 3.4.1 Beschreibung des Szenarios* beschriebenen Rahmenbedingungen ist der manuelle Ansatz (vgl. *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten*) die beste Wahl.

3.4.2 Szenario 2 – „Onkel Dagobert"

Das zweite Szenario ist typisch für die Einbindung einer Klassenbibliothek mit ca. 100 Klassen mit mind. je 20 Mitgliedern und verwandter Technologie. Es könnte sich um die Einbindung einer domänenspezifischen Programmierbibliothek – z. B. eine Bibliothek für Finanzmathematik – handeln.

Beschreibung des Szenarios

Es soll eine Bildverarbeitungsbibliothek mit 80 Klassen mit durchschnittlich jeweils 20 Mitgliedern und einer zentralen Bildklasse mit 100 Mitgliedern eingebunden werden. Die Klassenbibliothek ist in der gleichen Sprache implementiert wie das Datenflusssystem. Das Datenflusssystem unterstützt keine polymorphen Knotentypen. Es existiert keine Programmierschnittstelle, um die Metadaten der Klassenbibliothek zu verarbeiten. Die Metadaten liegen aber in Form einer XML-Datei vor.

Auswahl eines Lösungsansatzes

Da das Datenflussprogrammiersystem keine polymorphen Knotentypen unterstützt, kommt der Ansatz aus *Unterabschnitt 3.3.3 Polymorphe Brückenknoten* nicht in Frage. Die Anzahl der Interaktionen ist groß und die Metadaten liegen in maschinenlesbarer Form vor. Daher bietet sich die Implementierung eines Code-Generators nach *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten* an.

Der Implementierungsaufwand für einen Code-Generator rentiert sich bei einer großen Anzahl von Interaktionen, denn der Aufwand der Implementierung wird von der Anzahl der Interaktionen entkoppelt. Die Anzahl der generierten Knotentypen ist direkt abhängig von der Anzahl der Interaktionen und damit entsprechend groß. Das lässt sich aber mangels der Unterstützung für polymorphe Knotentypen nicht verhindern. Für die Ausweitung der Anbindung auf weitere Interaktionen oder bei einer Änderung der Interaktionen (z. B. durch eine Aktualisierung der Bildverarbeitungsbibliothek) ist evtl. ein erneutes Kompilieren der Brückenknotentypen erforderlich.

Bei Bedarf kann der generierte Quelltext der Brückenknoten für die einzelnen Interaktionen optimiert werden, wobei aber ein von der Anzahl der zu optimierenden Brückenknoten abhängiger Aufwand hinzukommt. Der Overhead zur Laufzeit fällt sehr niedrig aus, da die Bindungen an die Interaktionen statisch sind und kein Technologiebruch bei der Interaktion auftritt.

Für die in *Unterabschnitt 3.4.2 Beschreibung des Szenarios* beschriebenen Rahmenbedingungen ist der automatische Ansatz (vgl. *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten*) die beste Wahl.

3.4.3 Szenario 3 – „Käffchen"

Das dritte Szenario ist typisch, wenn eine ganze Programmiersprache an ein Datenflusssystem angebunden werden soll. Es könnte sich um die Anbindung von Klassenbibliotheken der Programmiersprache Java handeln.

Beschreibung des Szenarios

Es sollen alle Programmierbibliotheken einer Programmiersprache an ein Datenflusssystem angebunden werden. Das Datenflusssystem unterstützt polymorphe Knotentypen. Die anzubindende Programmiersprache ist technologiefremd mit dem Datenflusssystem. Es existiert eine Programmierschnittstelle für die Laufzeitumgebung der anzubindenden Programmiersprache, mit der Metadaten für die Programmierbibliotheken abgefragt und Interaktionen durchgeführt werden können.

Auswahl eines Lösungsansatzes

Es soll eine unbekannte Anzahl von Interaktionen eingebunden werden. Damit scheidet der manuelle Ansatz nach *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* aus. Da das Datenflusssystem polymorphe Knotentypen unterstützt, bietet sich der Ansatz mit polymorphen Brückenknoten an (vgl. *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*). Denn mit diesem Ansatz entfällt die Verwaltung von generiertem Quelltext (vgl. *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten*) und evtl. das Kompilieren beim Einbinden zusätzlicher Interaktionen.

Der Ansatz aus *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*, mit einem polymorphen Knotentyp je Interaktionsform, ist dem Spezialfall mit nur einem universellen Knotentyp i. d. R. vorzuziehen. Denn der Hauptvorteil des universellen polymorphen Brückenknotens, dass nur ein einziger Knoten in das Datenflusssystem eingebracht werden muss, fällt nicht so sehr ins Gewicht wie die erhöhte Komplexität der Implementierung eines solchen Knotentyps.

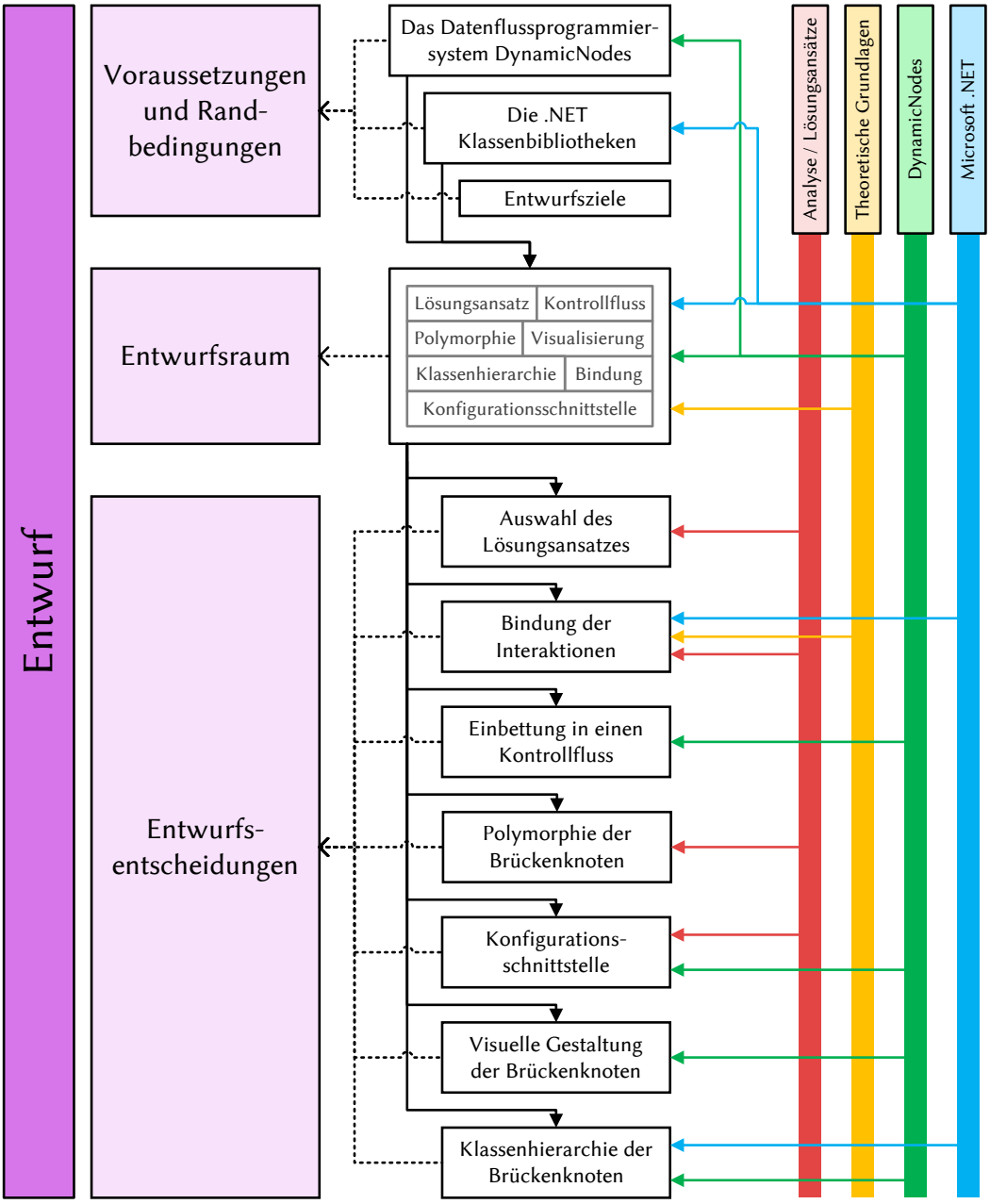
Der Implementierungsaufwand für die polymorphen Knotentypen ist unabhängig von der Anzahl der einzubindenden Interaktionen. Aus diesem Grund rentiert sich der erhöhte Aufwand für die Implementierung von polymorphen Knotentypen. Die Anzahl der Knotentypen entspricht der Anzahl der Interaktionsformen und ist damit klein im Vergleich zu der zu erwartendenden Anzahl von einzubindenden Interaktionen. Ein erneutes Kompilieren bei der Einbindung von zusätzlichen Interaktionen ist nicht erforderlich, damit steigt der Bedienkomfort für den Programmierer des Datenflusssystems.

Da die einzubindenden Interaktionen nicht bekannt sind, können die Knotentypen nicht für einzelne Interaktionen optimiert werden. Maschinenlesbare Metadaten sind über die Reflection-API erreichbar.

Der Overhead zur Laufzeit ist aus zwei Gründen recht hoch. Einerseits wird die Bindung der Interaktionen dynamisch durchgeführt und verbraucht damit in jeder Ausführungsphase der Brückenknoten Zeit. Andererseits existiert ein Technologiebruch zwischen dem Datenflusssystem und der anzubindenden Klassenbibliotheken, welcher ebenfalls für einen mehr oder weniger ausgeprägten Overhead sorgt.

Trotz der Nachteile im Bereich Performance und Anpassungsfähigkeit an einzelne Interaktionen ist für die in *Unterabschnitt 3.4.3 Beschreibung des Szenarios* beschriebenen Rahmenbedingungen der Ansatz „Polymorphe Brückenknoten je Interaktionsform" (vgl. *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*) die beste Wahl.

4 Entwurf


Entwurf
Voraussetzungen und Rand-bedingungen
Entwurfsraum
Entwurfs-entscheidungen
Das Datenflussprogrammier-system DynamicNodes
Die .NET Klassenbibliotheken
Entwurfsziele
Lösungsansatz
Kontrollfluss
Polymorphie
Visualisierung
Klassenhierarchie
Bindung
Konfigurationsschnittstelle
Auswahl des Lösungsansatzes
Bindung der Interaktionen
Einbettung in einen Kontrollfluss
Polymorphie der Brückenknoten
Konfigurations-schnittstelle
Visuelle Gestaltung der Brückenknoten
Klassenhierarchie der Brückenknoten
Analyse / Lösungsansätze
Theoretische Grundlagen
DynamicNodes
Microsoft .NET

Dieses Kapitel gliedert sich in drei Abschnitte. Aufbauend auf dem in *Kapitel 3 Analyse und abgeleitete Lösungsansätze* beschriebenen Konzept der Brückenknoten, soll für ein konkretes Szenario eine Brücke zwischen einem Datenflussprogrammiersystem und imperativen Klassenbibliotheken entworfen werden.

In Abschnitt 4.1 werden zunächst die Voraussetzungen für die Einbindung beschrieben. Dabei wird sowohl auf das Datenflussprogrammiersystem als auch auf die imperativen Klassenbibliotheken eingegangen. Es werden einige Besonderheiten der anzubindenden Klassenbibliotheken erläutert und anschließend ein passender Lösungsansatz aus *Abschnitt 3.3 Lösungsansätze* gewählt.

Darauf folgend, wird in Abschnitt 4.2 der Entwurfsraum beschrieben. Dabei wird der Entwurfsraum nicht vollständig behandelt. Es werden lediglich die wichtigsten Entwurfsentscheidungen herausgegriffen. In Abschnitt 4.3 werden die einzelnen Entwurfsentscheidungen getroffen. Dadurch wird die Grundlage für eine Implementierung gebildet.

4.1 Voraussetzungen und Randbedingungen

Um aus den in *Abschnitt 3.3 Lösungsansätze* entwickelten Lösungsansätzen einen passenden Ansatz auswählen zu können, müssen die Voraussetzungen bekannt sein, unter denen die Brücke entworfen werden soll. Die Voraussetzungen werden vor allem durch das ausgewählte Datenflussprogrammiersystem und die Klassenbibliotheken bestimmt, die in das Programmiersystem eingebunden werden sollen. Zu den Voraussetzungen des Entwurfs zählen auch Entwurfsziele, welche ebenfalls in diesem Abschnitt beschrieben werden.

4.1.1 Das Datenflussprogrammiersystem DynamicNodes

Als Datenflussprogrammiersystem wird DynamicNodes verwendet (vgl. [Kier08, Kier09a]). Das System ist ein visuelles Programmiersystem, welches selbst in C# implementiert ist und somit auf dem Microsoft .NET-Framework aufbaut. Es bietet zum jetzigen Entwicklungsstand eine kleine Knotenbibliothek für die Datenflusssteuerung, einige Knoten für allgemeine Aufgaben, wie die Ausgabe von Ergebniswerten und die Eingabe von Steuerwerten, und wenige weitere allgemeine Knoten. Der Schwerpunkt des Systems wird durch eine Knotenbibliothek zur Bildverarbeitung gebildet, die in ihrer Komplexität für Abiturienten und Studienanfänger in der Informatik oder den digitalen Medien geeignet ist.

In *Abbildung 4.1* ist die Benutzeroberfläche der Entwicklungsumgebung von DynamicNodes zu sehen.

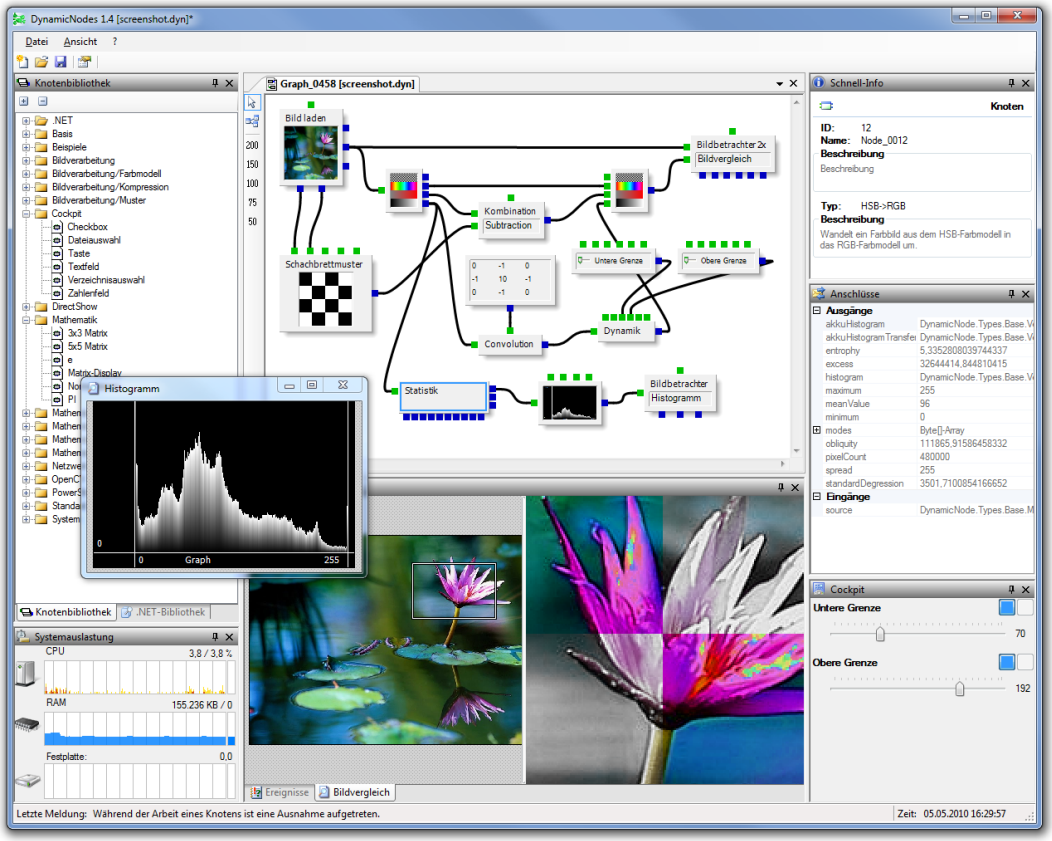


Abbildung 4.1: Die Entwicklungsumgebung von DynamicNodes

Aufbau der Knoten

Ein Knotentyp für DynamicNodes wird durch eine .NET-Klasse gebildet, die zumindest die Schnittstelle `DynamicNode.Core.INode`[1] implementiert oder aber (was wesentlich komfortabler ist) von einer der Basisklassen `DynamicNode.Core.Node`[2], `DynamicNode.Core.StreamNode`[3] oder `DynamicNode.Ext.Visual.VisualNode`[4] erbt.

DynamicNodes verwendet definierte Ein- und Ausgänge als Anschlüsse für Verbindungen. Ein Ausgang kann mit beliebig vielen Eingängen verbunden werden, ein Eingang nur mit genau einem Ausgang. Die Anschlüsse der Knoten in DynamicNodes besitzen eine Datentypgebundene Kompatibilität und bieten damit eine Unterstützung für typensicheres Programmieren innerhalb des Datenflussprogrammiersystems.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_INode.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_Node.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_StreamNode.htm`
[4] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Visual_VisualNode.htm`

Lebenszyklus der Knoten

Die Knoten in DynamicNodes werden als Instanzen einer Klasse erzeugt, welche die Schnittstelle `INode` implementiert. DynamicNodes ist ein statisches Datenflusssystem (vgl. *Struktur von Datenflussprogrammen* auf Seite 11) und erzeugt demzufolge beim Aufbau des Datenflussgraphen für jeden Knoten genau eine Instanz. Die Instanzierung der Knoten unterstützt keine Parametrisierung. Nach der Instanzierung ist jedoch in Abhängigkeit des jeweiligen Knotentyps eine vielfältige Konfiguration möglich.

Der Aufbau eines Datenflussgraphen kann entweder interaktiv, und über einen längeren Zeitraum, durch den Datenflussprogrammierer im Editor von DynamicNodes geschehen oder der Graph wird vollständig aus einer XML-Datei geladen.

Im Editor von DynamicNodes hat der Datenflussprogrammierer i. d. R. nur eine Möglichkeit einen Knoten in einen Graphen einzufügen (und dabei zu instanzieren): Eine Drag&Drop-Aktion von der Ansicht „Knotenbibliothek" auf die Arbeitsfläche des Editors. Dabei werden keine zusätzlichen Dialoge eingeblendet, welche eine Parametrisierung der Instanzierung ermöglichen, damit der Arbeitsfluss des Programmierers nicht gestört wird. Nachdem der Knoten jedoch in den Graphen eingefügt wurde, kann der Programmierer die Konfiguration des Knotens beliebig anpassen.

Während der Lebenszeit des Knotens kann der Datenflussgraph in einer XML-Datei gespeichert werden. Dabei wird auf jedem Knoten die Methode `void OnStore(IPropertyWriter)`[1] aufgerufen und eine leere Gruppe von Einstellungsparametern (`IPropertyWriter`[2]) übergeben. Dabei hat der Knoten die Möglichkeit seinen Zustand in Form von Einstellungsparametern zu speichern. Zusätzlich werden alle Verbindungen zwischen den Knoten in die XML-Datei geschrieben.

Wird ein Graph aus einer XML-Datei geladen, wird jeder Knoten zunächst instanziert und in den Graphen eingefügt. Anschließend wird die Methode `void OnRestore(IPropertyReader)`[3] auf dem Knoten aufgerufen. Dabei wird die Gruppe von Einstellungsparametern (`IPropertyReader`[4]), die in der XML-Datei für den jeweiligen Knoten hinterlegt ist, übergeben. Somit hat jeder Knoten die Möglichkeit seine Konfiguration nach der Instanzierung wiederherzustellen. Nachdem alle Knoten die Gelegenheit hatten, ihren Zustand zu rekonstruieren, werden alle in der XML-Datei gespeicherten Verbindungen zwischen den Knoten wieder aufgebaut.

[1] http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Persistence_IStorable_OnStore.htm
[2] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Persistence_IPropertyWriter.htm
[3] http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Persistence_IStorable_OnRestore.htm
[4] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Persistence_IPropertyReader.htm

Wird ein Graph entladen und zerstört, werden zunächst alle Verbindungen zerstört. Anschließend werden die Knoten aus dem Graphen entfernt und die Methode `void Dispose()`[1] auf ihnen aufgerufen. Durch diesen Aufruf erhalten die Knoten die Möglichkeit, Ressourcen, die sie während ihrer Lebenszeit in Anspruch genommen haben, wieder freizugeben.

Polymorphie

DynamicNodes unterstützt die Implementierung von polymorphen Knotentypen und polymorphen Knoten (vgl. *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen*). Das bedeutet, die Knoten können die polymorphe Initialisierungsphase jederzeit ausführen, um z. B. die Kompatibilität eines Anschlusses zu ändern. Sie können auch Anschlüsse entfernen und hinzufügen – sie unterstützen also voll polymorphe Knoten.

Kontrollfluss

DynamicNodes unterstützt die Verwendung eines Kontrollflusses in der Datenflussprogrammierung mit einigen allgemeinen Knoten und der Klasse `DynamicNode.Core.ControlTrigger`[2] (vgl. *Kontrollfluss in Datenflussgraphen* auf Seite 14). Die Klasse `ControlTrigger` implementiert das Design Pattern *Singleton* (vgl. [GaHeJoVl94]). Das ist deshalb sinnvoll, weil eine Instanz von `ControlTrigger` keine Informationen trägt und verschiedene Instanzen auch nicht unterschieden werden müssen. Durch das Singleton Pattern wird sichergestellt, dass der Speicher nicht durch viele unnötige Instanzen von `ControlTrigger` verbraucht wird. Anstelle dessen wird in allen Kontrollflüssen innerhalb DynamicNodes dieselbe Instanz verwendet. Diese eine Instanz wird über die statische Eigenschaft `ControlTrigger.Instance`[3] abgerufen.

Ein Knoten, der auf einen Kontrollfluss reagieren kann, besitzt einen Eingang mit dem Datentyp `System.Object`[4]. Damit kann er nicht nur die Steuermarken vom Typ `ControlTrigger` entgegennehmen, sondern auch beliebige andere Verbindungen als Steuersignal verarbeiten. Das ist möglich, weil alle Datentypen im .NET-Framework von `System.Object` abgeleitet sind. Ein Knoten, der einen Kontrollfluss weitergeben kann, besitzt einen Ausgang mit dem Datentyp `ControlTrigger`. Wenn ein Steuersignal über diesen Ausgang weitergegeben werden soll, muss der Knoten eine Marke mit dem Markenwert `ControlTrigger.Instance` auf den Ausgang platzieren.

[1] http://msdn.microsoft.com/de-de/library/system.idisposable.dispose.aspx
[2] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_ControlTrigger.htm
[3] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Core_ControlTrigger_Instance.htm
[4] http://msdn.microsoft.com/en-us/library/system.object.aspx

Benutzeroberfläche

DynamicNodes ist ein visuelles Programmiersystem. Ein Schwerpunkt der Entwicklungsumgebung liegt auf dem grafischen Editor, in dem die Datenflussgraphen modelliert werden. In diesem Editor erscheint jeder Knoten mit seinen Anschlüssen.

DynamicNodes erzeugt für Knotentypen, welche keinerlei Vorgaben für die grafische Darstellung machen, automatisch eine Standardvisualisierung (vgl. *Abbildung 4.2*). Dabei geht DynamicNodes nach dem folgenden Schema vor: Die Knoten werden als Rechteck mit 3D-Kanten im Stil der Benutzeroberfläche von Windows 2000 dargestellt. Alle Eingänge des Knotens werden an der linken Seite des Knotens in der Reihenfolge der Erzeugung angeordnet. Die Ausgänge werden auf die gleiche Weise an der rechten Seite des Knotens angeordnet. Der Name des Knotentyps erscheint linksbündig oben innerhalb des Rechtecks. Die Höhe des Knotens ergibt sich aus dem Maximum von der Anzahl der Eingänge und der Anzahl der Ausgänge. Die Breite ergibt sich aus der Länge des Namens.

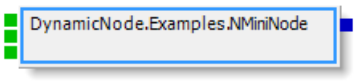


Abbildung 4.2: Standardvisualisierung eines Knotens

Ein Knotentyp kann eine rechteckige Darstellungsfläche innerhalb der Visualisierung des Knotens vollständig selbst zeichnen, um Symbole oder Text anzuzeigen, der die Funktion des Knotens oder seinen Zustand visualisiert. In *Abbildung 4.3* ist ein Knoten als Beispiel zu sehen, der den Datenfluss, durch eine Bedingung gesteuert, verzweigt. Dieser Knoten stellt seinen Schaltzustand durch ein entsprechendes Symbol dar. Seinen Namen hat dieser Knotentyp ausgeblendet, auch hat er die Position der Anschlüsse selbst definiert.

Abbildung 4.3: Ein Knoten, der seinen Zustand visualisiert

Knoten können in DynamicNodes eine sog. Knotensteuerung anbieten. Dabei handelt es sich um eine grafische Benutzerschnittstelle, welche direkt mit dem Knoten verbunden ist und eine Steuerung oder Konfiguration des Knotens ermöglicht. DynamicNodes bietet dazu in der Benutzeroberfläche eine Ansicht mit dem Namen „Knotensteuerung". Wird ein Knoten auf der Arbeitsfläche ausgewählt, wird seine Knotensteuerung – sofern er eine besitzt – in der gleichnamigen Ansicht angezeigt. In *Abbildung 4.4* ist die Knotensteuerung von einem Knoten zu sehen, der dem Benutzer erlaubt eine 3x3-Matrix mit Werten zu füllen.

Abbildung 4.4: Die Knotensteuerung des Knotens „Matrix 3x3"

4.1.2 Die .NET-Klassenbibliotheken

Die Einsatzmöglichkeiten des in *Unterabschnitt 4.1.1 Das Datenflussprogrammiersystem DynamicNodes* beschriebenen Systems werden erheblich erweitert, wenn es gelingt eine Brücke für die Einbindung von beliebigen .NET-Klassenbibliotheken zu schaffen. Als .NET-Klassenbibliotheken werden alle DLL- und EXE-Dateien bezeichnet, die auf der CLR und dem CTS des .NET-Frameworks basieren. Diese Klassenbibliotheken werden im Microsoft-Umfeld als *Assemblies* bezeichnet (vgl. *Abschnitt 2.3 Microsoft .NET-Framework*).

Das Potential einer Brücke zwischen DynamicNodes und .NET-Klassenbibliotheken liegt vor allem in zwei Punkten:

1. Die Klassenbibliotheken, die im .NET-Framework (Version 3.5) selbst enthalten sind, umfassen 35.863 öffentliche Typen, davon 24.903 Klassen. Alle Typen zusammen (Klassen, Schnittstellen, Strukturen, u.a.) besitzen 1.007.781 öffentliche Mitglieder (vgl. *Abschnitt A.2 Benchmark für Bindungstechniken* im Anhang auf Seite 145). Die Themengebiete, die durch diese Klassenbibliotheken abgedeckt werden, umfassen einfache Mathematik, Verarbeitung von Zeichenketten, Eingabe/Ausgabe, Verarbeitung von XML, Benutzeroberflächen, Kommunikation über Netzwerk, Anbindung an relationale Datenbanken und viele mehr (vgl. [Micr10b]).
2. Es gibt unzählige Projekte, deren Ergebnisse in Form von .NET-Klassenbibliotheken verfügbar sind. Durch sie werden weitere Themen abgedeckt, die durch die Klassenbibliotheken des .NET-Frameworks nicht abgedeckt werden.

4.1.3 Entwurfsziele

Mit dem Entwurf einer Brücken zwischen DynamicNodes und den .NET-Klassenbibliotheken sollen die folgenden Entwurfsziele verfolgt werden:

- Niedriger Implementierungsaufwand
- Hohe Performance
- Selbsterklärende Benutzeroberfläche

- Einfache Bedienbarkeit

Mit dem Ziel „Niedriger Implementierungsaufwand" ist primär die Anzahl von erforderlichen Code-Zeilen, sekundär auch die Informationsdichte des notwendigen Quellcodes gemeint. Das bedeutet, dass eine große Menge an Quellcode auch einen hohen Aufwand erfordert, häufige Wiederholungen innerhalb des Quellcodes den Aufwand aber senken.

Mit dem Ziel „Hohe Performance" ist gemeint, dass die Brückenknoten einen möglichst geringen Overhead an Laufzeit erzeugen sollen.

Das Ziel einer „Selbsterklärenden Benutzeroberfläche" verfolgt den Entwurf einer Benutzeroberfläche, die ein .NET-kundiger Programmierer ohne Zuhilfenahme von zusätzlicher Dokumentation bedienen kann.

Das Ziel „Einfache Bedienbarkeit" verfolgt den Entwurf einer Benutzeroberfläche, mit der die Einbindung von .NET-Klassenbibliotheken in einen Datenflussgraph mit einer geringen Anzahl an Maus-Klicks möglich ist. „Einfach" bedeutet hier also auch „effizient" aus Sicht des Datenflussprogrammierers.

Da sich in vielen Situationen der Softwareentwicklung die Ziele „Niedriger Implementierungaufwand" und „Hohe Performance" widersprechen, ist das wichtigste Entwurfsziel ein Kompromiss aus niedrigem Implementierungsaufwand und hoher Performance. Diese Arbeit möchte das Konzept der Brückenknoten mit einer Beispielimplementierung veranschaulichen, aus diesem Grund sollte der Kompromiss eher zu Lasten der Performance gebildet werden. Die Performance darf jedoch nicht vollständig außer Acht gelassen werden, damit die entwickelte Brücke einen nachhaltigen Nutzwert für das DynamicNodes-System erhält. Kann ein wesentlicher Performance-Gewinn nur mit einer aufwändigen Implementierung erzielt werden, ist demzufolge eine einfache Implementierung vorzuziehen.

Das Ziel für die Gestaltung der grafischen Benutzeroberfläche der Knoten ist, dass sie selbsterklärend ist und die Bedienung einfach sein soll. Der Aufwand für die Gestaltung der Benutzeroberfläche soll möglichst niedrig gehalten werden.

4.2 Entwurfsraum

In diesem Abschnitt wird der wesentliche Teil des Entwurfsraums für den Entwurf der Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken dargestellt.

Die Erläuterung aller Entwurfsentscheidungen, die zu einer konkreten Implementierung führen, würde den Rahmen dieser Arbeit bei weitem sprengen. Deshalb wird der Entwurfsraum nicht vollständig beschrieben. Es werden nur die Entwurfsentscheidungen behandelt, welche nach dem Ermessen des Autors die Wichtigsten darstellen.

Lösungsansätze

Als Ausgangspunkt für den Entwurf muss ein Lösungsansatz aus *Abschnitt 3.3 Lösungsansätze* gewählt werden. Als Vorauswahl werden für den Entwurfsraum all jene Lösungsansätze ausgewählt, die zu den Voraussetzungen passen. Die einzige Einschränkung bei den Lösungsansätzen kann sich bei dem dritten Ansatz „Polymorphe Brückenknoten" ergeben, falls das Datenflusssystem keine polymorphen Knotentypen unterstützt. DynamicNodes unterstützt voll polymorphe Knoten(typen), demzufolge können alle Lösungsansätze für den Entwurfsraum ausgewählt werden. Das sind die Ansätze „Manuell implementierte Brückenknoten", „Automatisch generierte Brückenknoten" und „Polymorphe Brückenknoten".

Bindungstechniken

Für die Ausführung von Interaktionen ist eine Bindung erforderlich (vgl. *Bindung* auf Seite 31). Als Bindung kommen drei Möglichkeiten in Frage:

1. **Direkte Bindung** durch hart kodierte Adressierung von Interaktionen
   Diese Bindung kommt für die Lösungsansätze „Manuell implementierte Brückenknoten" und „Automatisch generierte Brückenknoten" in Frage.
2. **Feste Bindung** durch Erzeugung von Bindungs-Code zur Laufzeit (vgl. *IL-Emitter / Lambda-Ausdrücke / C#-Compiler* auf Seite 28)
   Diese Bindung kommt für den Lösungsansatz „Polymorphe Brückenknoten" in Frage.
3. **Lose Bindung** durch indirekte Ausführung mit Hilfe der Reflection-API (vgl. *Reflection-API* auf Seite 27)
   Diese Bindung kommt für den Lösungsansatz „Polymorphe Brückenknoten" in Frage.

Einbettung in einen Kontrollfluss

Da DynamicNodes einen Kontrollfluss unterstützt, muss für die Brückenknoten entschieden werden, ob und wie sie einen Kontrollfluss unterstützen. Dabei müssen die Anschlüsse und das Verhalten der Brückenknoten entsprechend den Interaktionsformen definiert werden.

Formen der Polymorphie

Falls die Brückenknoten polymorph implementiert werden sollen, muss entschieden werden, ob sie schwach polymorph oder voll polymorph implementiert werden sollen (vgl. *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen*).

Die Polymorphie von Operationsknoten kann laut *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen* entweder ausschließlich durch eine Initialisierungsphase während der

Instanzierung (polymorphe Knotentypen) oder durch eine wiederholt ausführbare Initialisierungsphase nach der Instanzierung (polymorphe Knoten) realisiert werden. Die Initialisierung muss bei jeder Ausführung parametrisiert werden, um die Polymorphie zu ermöglichen. Soll die Initialisierungsphase während der Instanzierung durchgeführt werden muss das datenflussorientierte Programmiersystem eine parametrisierte Instanzierung von Knoten unterstützen.

Von DynamicNodes wird keine parametrisierte Instanzierung unterstützt (vgl. *Unterabschnitt 4.1.1 Das Datenflussprogrammiersystem DynamicNodes*), deshalb muss die Initialisierungsphase zur Realisierung der Polymorphie nach der Instanzierung ausgeführt werden.

Benutzerschnittstelle zur Konfiguration der Polymorphie

Falls die Brückenknoten polymorph implementiert werden sollen, muss dem Datenflussprogrammierer eine Benutzerschnittstelle zur Verfügung gestellt werden, welche die Parametrisierung der initialisierungsphase ermöglicht. Diese Benutzerschnittstelle muss ggf. entworfen werden.

Visualisierung der Brückenknoten

DynamicNodes ist ein visuelles Programmiersystem, in dem jeder Knotentyp seine Visualisierung selbst steuern kann. Auch die Brückenknoten sollen die Standardvisualisierung, die DynamicNodes zur Verfügung stellt, anpassen, um dem Programmierer die Arbeit zu erleichtern. Die folgenden Möglichkeiten bieten sich: Die Position der Anschlüsse kann angepasst werden. Der Name des Knotens kann ausgeblendet werden. Der Knoten kann eine rechteckige Darstellungsfläche einblenden und selbst zeichnen. Diese Darstellung muss ggf. entworfen werden.

Klassenhierarchie

Als letzten Entwurfsschritt muss eine Klassenhierarchie entworfen werden, welche die Gemeinsamkeiten zwischen den Brückenknoten in Basisklassen zusammenfasst. Dazu muss zunächst eine Basisklasse des DynamicNodes-Systems als Wurzel für die Hierarchie ausgewählt werden. Dazu stehen die folgenden Basisklassen zur Auswahl: Eine eigene Klasse, welche die Schnittstelle `DynamicNode.Core.INode`[1] implementiert, `DynamicNode.Core.Node`[2], `DynamicNode.Core.StreamNode`[3] oder `DynamicNode.Ext.Visual.VisualNode`[4].

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_INode.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_Node.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_StreamNode.htm`
[4] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Visual_VisualNode.htm`

4.3 Entwurfsentscheidungen

In diesem Abschnitt werden die einzelnen Entwurfsentscheidungen für die Entwicklung der Brückenknoten, unter Berücksichtigung des Entwurfsraums, gefällt.

4.3.1 Auswahl des Lösungsansatzes

Die Voraussetzungen in *Abschnitt 4.1 Voraussetzungen und Randbedingungen* ermöglichen die Auswahl eines Lösungsansatzes aus *Abschnitt 3.3 Lösungsansätze*. Da eine unbekannte und potentiell große Anzahl an Interaktionen in DynamicNodes eingebunden werden soll, kommt ein manueller Ansatz wie in *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten* nicht in Frage.

Da DynamicNodes polymorphe Knoten unterstützt, muss als nächstes zwischen einem automatischen Ansatz mit Code-Generator (vgl. *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten*) und einem Ansatz mit polymorphen Knotentypen (vgl. *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*) gewählt werden. Die polymorphen Knotentypen haben den großen Vorteil, dass beim Einbinden von weiteren Interaktionen kein erneutes Kompilieren der Brückenknoten erforderlich ist. Des Weiteren ist die Anzahl der Knotentypen, die durch das Datenflussprogrammiersystem verwaltet werden müssen auf die Anzahl der Interaktionsformen begrenzt. Aus diesen beiden Gründen wird ein Ansatz mit polymorphen Brückenknoten verfolgt.

Abschließend steht zur Auswahl, ob ein universeller polymorpher Brückenknoten oder ein polymorpher Brückenknoten je Interaktionsform verwendet werden soll. Da der Vorteil eines einzelnen Brückenknotens den Nachteil der erhöhten Komplexität nicht aufwiegt, wird der Ansatz der polymorphen Brückenknoten je Interaktionsform gewählt.

4.3.2 Bindung der Interaktionen

Die Aufgabe der Brückenknoten ist es, eine Interaktion mit einer imperativen Klassenbibliothek zu ermöglichen.

In diesem Unterabschnitt wird zunächst eine Technik zur Adressierung von Interaktionen innerhalb des .NET-Frameworks entworfen. Diese Adressen sind Voraussetzung für eine Bindung durch die Brückenknoten. Anschließend werden die Bindungstechniken behandelt.

Da die Brückenknoten polymorph sind, muss die Bindung der Interaktionen, z. B. ein Methodenaufruf mit Parametern, zur Laufzeit erfolgen. Die direkte Bindung im Quelltext kommt nicht in Frage, da die Brückenknoten polymorph sind.

Der Vorteil der technologischen Verwandtschaft, welche in *Bindung* auf Seite 31 beschrieben sind, kommen in diesem Fall zum Tragen, da die Knoten von DynamicNodes selbst auf dem .NET-Framework basieren.

Für die Brückenknoten zwischen DynamicNodes und den .NET-Klassenbibliotheken stehen zwei Möglichkeiten zur Auswahl.

1. **Feste Bindung** durch Generierung von IL-Verbindungs-Code zur Laufzeit und Aufruf mit über Delegaten (vgl. *IL-Emitter / Lambda-Ausdrücke / C#-Compiler* auf Seite 28).
2. **Lose Bindung** durch indirekten Aufruf der zu bindenden Interaktion mit Hilfe der Reflection-API (vgl. *Reflection-API* auf Seite 27).

Beide Techniken werden genauer beschrieben. Es werden ihre Vor- und Nachteile erörtert. Darauf aufbauend wird eine Entscheidung für eine der beiden Techniken gefällt.

Adressierung von Klassen und Klassenmitgliedern

In *Unterabschnitt 3.2.2 Das Konzept der Brückenknoten für die Einbindung von imperativen Klassenbibliotheken* wird von einer Brücke zu einer imperativen, objektorientierten Klassenbibliothek gefordert, dass sie die Auswahl der Adresse von Klassen und von Klassenmitgliedern erlauben muss. Diese Adressen bilden die Grundlage für die Bindung der Klassenmitglieder zur Interaktion.

Das .NET-Framework stellt diese Adressen als Teil der Reflection-API bereit (vgl. *Reflection-API* auf Seite 27), ohne echte Speicher- oder Einsprungadressen zu veröffentlichen. Als Adresse einer Klasse dienen Objekte vom Typ `System.Type`[1]. Als Adresse für Klassen-Mitglieder dienen Objekte vom Typ `System.Reflection.MemberInfo`[2].

Um dem Programmierer des Datenflusssystems eine Auswahl von Klassen und Schnittstellen zur Verfügung zu stellen, ist es möglich, alle Typen der geladenen Assemblies zu ermitteln und die Klassen und Schnittstellen herauszufiltern. Die Typen sind durch die Assemblies gruppiert. Um alle Typen zu ermitteln, müssen zunächst alle geladenen Assemblies ermittelt werden. Die Assemblies werden für die Ausführung in einem .NET-Prozess in sog. *Application Domains* geladen. Eine Application Domain bildet einen abgeschlossen Raum für die Speicherverwaltung.

Um alle geladenen Assemblies zu ermitteln, wird ein `System.AppDomain`[3]-Objekt benötigt, welches die aktuelle Application Domain beschreibt. Dieses Objekt besitzt eine Methode `Assembly[] GetAssemblies()`[4], die ein Array mit Assembly-Objekten für alle geladenen As-

[1] `http://msdn.microsoft.com/en-us/library/system.type.aspx`
[2] `http://msdn.microsoft.com/en-us/library/system.reflection.memberinfo.aspx`
[3] `http://msdn.microsoft.com/en-us/library/system.appdomain.aspx`
[4] `http://msdn.microsoft.com/en-us/library/system.appdomain.getassemblies.aspx`

semblies zurückgibt. Für jede Assembly lässt sich nun mit der Methode `Type[] GetTypes()`[1] ein Array mit allen öffentlichen Typen der Assembly erfragen. Die Menge der `Type`-Objekte lässt sich anschließend nach Schnittstellen und Klassen filtern. *Listing 4.1* veranschaulicht dieses Vorgehen unter Verwendung von LINQ-Syntax (verfügbar ab .NET-Framework 3.5).

Listing 4.1: Beispiel für die Nutzung der Reflection-API (Teil 1)

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Reflection;
5
6  namespace de.mastersign.demo
7  {
8      class ReflectionDemo
9      {
10         public IEnumerable<Type> GetAllTypes()
11         {
12             var domain = AppDomain.CurrentDomain;
13             return
14                 from assembly in domain.GetAssemblies()
15                 from type in assembly.GetTypes()
16                 where type.IsInterface || type.IsClass
17                 select type;
18         }
```

Die Auflistung aller Klassenmitglieder zur Auswahl ist mit der Methode `MemberInfo[] GetMembers()`[2] der Klasse `Type` möglich.

Listing 4.2: Beispiel für die Nutzung der Reflection-API (Teil 2)

```
20         public IEnumerable<MemberInfo> GetAllMembers()
21         {
22             return
23                 from type in GetAllTypes()
24                 from member in type.GetMembers()
25                 select member;
26         }
27     }
28 }
```

[1] http://msdn.microsoft.com/en-us/library/system.reflection.assembly.gettypes.aspx
[2] http://msdn.microsoft.com/en-us/library/424c79hc.aspx

Lose Bindung mit Hilfe der Reflection-API

Mit Hilfe der Reflection-API (vgl. *Reflection-API* auf Seite 27) ist es einfach möglich, Interaktionen auszuführen, die zur Kompilierzeit noch unbekannt sind und zur Laufzeit durch einen Namen oder durch Aufzählung aller vorhandenen Interaktionen ermittelt wurden. Soll z. B. eine statische Methode von einer Klasse aufgerufen werden, wobei die Adresse der Klasse und der Methode nur als Zeichenkette vorliegt, kann der Code in *Listing 4.3* zum Aufruf verwendet werden. Obwohl der Beispiel-Code an dieser Stelle keine Parameter oder Rückgabewerte unterstützt, ist auch das einfach möglich.

Listing 4.3: Aufruf einer statischen Methode mit der Reflection-API

```
1  using System;
2  namespace de.mastersign.demo
3  {
4      class ReflectionHelper
5      {
6          public void CallStaticMethod(string className, string
               methodName)
7          {
8              var type = Type.GetType(className);
9              var method = type.GetMethod(methodName);
10             var parameterValues = new object[] {};
11             method.Invoke(null, parameterValues);
12         }
13     }
14 }
```

Für die Demonstration der Bindung einer statischen Methode wird hier in Zeile 9 ein Objekt der Klasse `System.Reflection.MethodInfo` ermittelt (`method`) und in Zeile 11 verwendet. Für die übrigen Mitglieder von Klassen existieren vergleichbare Klassen (`FieldInfo`, `PropertyInfo`, `ConstructorInfo`, etc.)

Feste Bindung mit dynamisch erzeugtem Verbindungs-Code

Eine andere Möglichkeit der Bindung ist das dynamische Erzeugen von Verbindungs-Code. Dabei wird die Möglichkeit ausgenutzt, zur Laufzeit IL-Code zu erzeugen und in den aktuellen Prozess derart zu emittieren, dass er anschließend sofort ausgeführt werden kann (vgl. *IL-Emitter / Lambda-Ausdrücke / C#-Compiler* auf Seite 28).

Die Interaktion wird bei der festen Bindung von einer dynamisch erzeugten Methode ausgeführt, die in der polymorphen Initialisierungsphase des Brückenknotens kompiliert wird. Für einen Brückenknoten, der einen statischen Methodenaufruf durchführt, bedeutet das, dass die

Parameter und der Rückgabewert der einzubindenden Methode verwendet werden, um eine Methode zu erzeugen, welche die Signatur `Object Invoke(Object[] parameters)` besitzt. Diese Methode ruft intern direkt die einzubindende statische Methode auf und leitet ihr die, in dem Array `parameters` übergebenen, Parameter weiter. Gibt die aufgerufene statische Methode einen Rückgabewert zurück, wird dieser auch an den Aufrufer der dynamischen Methode zurückgegeben, andernfalls wird `null` zurückgegeben.

Durch diese direkte Bindung, mit Hilfe des zur Laufzeit erzeugten IL-Codes, verschmilzt die einzubindende Klassenbibliothek mit den Brückenknoten, als wäre die eingebundene Klassenbibliothek schon zur Kompilierzeit der Brückenknoten bekannt gewesen und direkt im Quellcode verwendet worden.

Die feste Bindung vereint einige Vorteile des Lösungsansatzes mit automatisch generiertem Quellcode (vgl. *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten*) mit Vorteilen des Lösungsansatzes mit polymorphen Knoten (vgl. *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*). Aus dem Lösungsansatz mit automatisch generiertem Quellcode wird die gute Performance übernommen, jedoch nicht die Anpassungsfähigkeit je Interaktion. Aus dem Lösungsansatz mit polymorphen Knoten wird der Vorteil der geringen Anzahl an Brückenknotentypen übernommen. Die Komplexität der Implementierung ist im Vergleich zur Implementierung von polymorphen Brückenknoten mit loser Bindung höher.

Auswahl einer Bindungstechnik

Damit eine der beiden Bindungstechniken ausgewählt werden kann, müssen die Vor- und Nachteile jeder Technik betrachtet werden.

Der Vorteil einer losen Bindung ist ein einfacher Aufbau der Implementierung für die Polymorphie der Brückenknoten. Ist z. B. ein `MethodInfo`-Objekt als Adresse zu einer Methode ausgewählt, erfolgt die Bindung durch einen einfachen Aufruf von `MethodInfo.Invoke()`.

Der Nachteil der losen Bindung ist die schlechte Performance (vgl. *Abschnitt A.2 Benchmark für Bindungstechniken* im Anhang auf Seite 145), da eine indirekte Interaktion mit Klassenmitgliedern durch die Reflection-API einen großen Overhead erzeugt.

Der Vorteil der festen Bindung ist wiederum die Performance. Denn der Aufruf der zur Laufzeit kompilierten Methode für die Interaktion mit Hilfe eines Delegaten ist einem manuell oder automatisch implementierten Brückenknoten mit direktem Aufruf der Interaktion ebenbürtig (vgl. *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten*, *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten* und *Abschnitt A.2 Benchmark für Bindungstechniken* im Anhang auf Seite 145).

Der Nachteil der festen Bindung ist die größere Komplexität gegenüber der losen Bindung. Unter Ausnutzung der, in .NET 3.5 neuen, API zur Erzeugung und Kompilierung von Lambda-

Ausdrücken (vgl. *IL-Emitter / Lambda-Ausdrücke / C#-Compiler* auf Seite 28) ist die Komplexität zwar akzeptabel, jedoch sind die Brückenknoten dann nur auf dem .NET-Framework in der Version 3.5 lauffähig, wohingegen das Programmiersystem DynamicNodes bisher noch mit dem Funktionsumfang des .NET-Frameworks in der Version 2.0 auskommt.

Auf dem .NET-Framework in der Version 2.0 ist die Erzeugung der Bindungsmethoden nur unter Verwendung von IL-Befehlen (Assembler ähnliche OpCodes) möglich (vgl. *IL-Emitter / Lambda-Ausdrücke / C#-Compiler* auf Seite 28). Der Einsatz von IL-Befehlen bei der Implementierung der Brückenknoten hat jedoch zur Folge, dass sich der entsprechende Quelltext dem Verständnis der meisten C#-Programmierer entzieht und deshalb schlecht zu warten ist.

In den Entwurfszielen wurde bestimmt, dass eine performantere Implementierung nur gewählt werden soll, wenn die Komplexität der Implementierung dadurch nicht unverhältnismäßig ansteigt (vgl. *Unterabschnitt 4.1.3 Entwurfsziele*). Aus diesem Grund wird der losen Bindung der Vorzug gegeben.

Als Anmerkung sei jedoch gesagt, dass für einen produktiven Einsatz der Brückenknoten, in dem die Performance eine größere Rolle spielt, die feste Bindung umgesetzt werden sollte.

4.3.3 Anschlüsse für die Einbettung in einen Kontrollfluss

DynamicNodes unterstützt die Verwendung eines Kontrollflusses in der Datenflussprogrammierung mit einigen allgemeinen Knoten und dem Datentyp `DynamicNode.Core.ControlTrigger`[1] (vgl. *Kontrollfluss* auf Seite 54). In diesem Unterabschnitt wird beschrieben, wie die polymorphen Brückenknoten den Kontrollfluss unterstützen.

Eingänge für einen Kontrollfluss

Die Herausforderung bei der Einbettung eines Brückenknotens in einen Kontrollfluss ist, den Knoten nicht von dem Kontrollfluss abhängig zu machen. Angenommen, es wird ein Eingang *triggerIn* für Steuermarken definiert, dann sollte der Brückenknoten nur aktiviert werden, wenn über den Eingang *triggerIn* eine Marke einläuft. Laufen auf weiteren Eingängen Daten ein, z. B. neue Parameterwerte für einen Methodenaufruf, sollte der Knoten nicht aktiviert werden.

Wird der Knoten nun aber ohne expliziten Kontrollfluss verwendet, so sollte der Knoten dem normalen Aktivierungsmuster folgen und aktiviert werden, sobald an einem beliebigen seiner Eingänge eine neue Marke einläuft (vgl. [Kier08, S. 45-47]). Dies ist jedoch nicht der Fall, da der Brückenknoten nur durch einlaufende Marken über *triggerIn* aktiviert wird. Eine Notlösung wäre, einen der Ausgängen (*xOut*), welche Daten für die Eingänge des Brückenknotens

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_ControlTrigger.htm`

liefern, mit *triggerIn* zu verbinden. Dadurch würde der Knoten aktiviert werden, wenn neue Daten von *xOut* eintreffen. Es können aber nicht mehrere Ausgänge mit *triggerIn* verbunden werden. Es ist also auch auf Umwegen nicht möglich, das korrekte Verhalten herbeizuführen, wenn *triggerIn* die einzige Aktivierungsmöglichkeit für den Brückenknoten ist.

Es liegen nun zwei unterschiedliche Aktivierungsmuster für die Brückenknoten vor, die durch ein Kriterium gesteuert abwechselnd in Kraft treten müssen. Das eine Muster ist das gewöhnliche Verhalten eines DynamicNodes-Knotens und das Zweite die Aktivierung ausschließlich durch einlaufende Marken über *triggerIn*.

Das Kriterium für die Aktivierung der Muster ist: „Wird der Brückenknoten in einem Kontrollfluss verwendet?" Ist dieses Kriterium nicht erfüllt, soll das erste Aktivierungsmuster in Kraft treten, ist es erfüllt, soll das Zweite in Kraft treten. Das Kriterium kann auf einfache Weise auf die Programmierung des Brückenknotens abgebildet werden. Dazu muss lediglich überprüft werden, ob der Eingang *triggerIn* mit einem Ausgang verbunden ist.

Für diesen Zweck bietet die Schnittstelle `DynamicNode.Core.IPort`[1], welche von der Klasse `DynamicNode.Core.InPort`[2] implementiert wird, die Eigenschaft `IPort.ConnectionCount`[3]. Für einen Eingang kann diese Eigenschaft nur die Werte 0 oder 1 annehmen. Ist `ConnectionCount` von *triggerIn* gleich 1, dann wird der Brückenknoten in einem Kontrollfluss verwendet und darf nur durch *triggerIn* aktiviert werden. Ist `ConnectionCount` von *triggerIn* gleich 0, wird der Brückenknoten ohne einen Kontrollfluss verwendet und muss dem normalen Aktivierungsmuster folgen.

Alle Brückenknoten sollten einen Eingang *triggerIn* bieten, um die Aktivierung des Knotens durch einen Kontrollfluss steuern zu können. Nur zwei Interaktionsformen bilden hier eine Ausnahme: das Überwachen von objektgebundenen Ereignissen und das Überwachen von statischen Ereignissen.

Beim Überwachen von einem Ereignis gibt es zwei unterschiedliche Phasen der Aktivierung des Brückenknotens. Die erste Phase wird durch das Eintreffen einer Marke mit einem neuen Objektkontext an dem Eingang *instance* ausgelöst. Dabei wird der neue Objektkontext entgegengenommen und die Überwachung des Ereignisses gestartet bzw. beendet. Diese Phase gibt es nur bei der Überwachung von objektgebundenen Ereignissen. Bei der Überwachung von statischen Ereignissen findet das Starten und Stoppen in der Initialisierungsphase des Brückenknotens statt. Die zweite Phase wird durch das überwachte Ereignis ausgelöst. Während dieser Aktivierung werden die Ereignisdaten über Ausgänge weitergeleitet.

Durch einen Eingang für den Kontrollfluss könnte nur die erste Aktivierungsphase kontrolliert werden, da die zweite durch das überwachte Ereignis kontrolliert wird. Das erscheint

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_IPort.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_InPort.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Core_IPort_ConnectionCount.htm`

wenig sinnvoll. Aus diesem Grund benötigen die Brückenknoten zur Überwachung von Ereignissen keinen Eingang *triggerIn*.

Ausgänge für einen Kontrollfluss

Alle Brückenknoten sollten auch einen Ausgang *triggerOut* bieten, um einen Kontrollfluss starten oder weiterleiten zu können. Immer wenn ein Brückenknoten die Ausführung einer Interaktion beendet hat, muss über diesen Ausgang eine Marke mit dem Wert `ControlTrigger.Instance` weitergegeben werden.

Bei Brückenknoten, die Ereignisse überwachen, wird über den Ausgang *triggerOut* nur dann eine Marke weitergegeben, wenn das überwachte Ereignis ausgelöst wurde.

4.3.4 Polymorphie der Brückenknoten

Das Datenflussprogrammiersystem DynamicNodes unterstützt polymorphe Knotentypen, die sowohl während der Instanzierung als auch zu einem späteren Zeitpunkt ihre Anschlüsse anpassen dürfen (vgl. *Unterabschnitt 4.1.1 Das Datenflussprogrammiersystem DynamicNodes*). Knotentypen werden als .NET-Klasse definiert, welche die Schnittstelle `DynamicNode.Core.INode`[1] implementieren und Anschlüsse sind Objekte der Klasse `DynamicNode.Core.InPort`[2] oder `DynamicNode.Core.OutPort`[3].

Erzeugung der Anschlüsse

DynamicNodes unterstützt zwei verschiedene Arten, um Anschlüsse für einen Knoten zu erzeugen. Es muss entschieden werden, welche Art für die Brückenknoten geeignet ist. Nicht polymorphe Knotentypen erzeugen ihre Anschlüsse im Konstruktor (vgl. *Listing 4.4*) oder über *Annotationen* an öffentlichen Feldern oder Eigenschaften (vgl. *Listing 4.5*). Beide Varianten haben Vor- und Nachteile. Es muss geklärt werden, welche Art der Erzeugung von Anschlüssen für polymorphe Knoten geeignet ist.

Listing 4.4: Erzeugung von Anschlüssen durch Instanzierung

```
1  using DynamicNode.Core;
2
3  namespace de.mastersign.demo
4  {
5      public class NonPolymorphA : Node
6      {
```

[1] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_INode.htm
[2] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_InPort.htm
[3] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_OutPort.htm

```
7         private InPort aIP;
8         private InPort bIP;
9         private OutPort resultOP;
10
11        public NonPolymorphA()
12        {
13            aIP = new InPort(this, "a", typeof(int));
14            bIP = new InPort(this, "b", typeof(int));
15            resultOP = new OutPort(this, "result", typeof(int));
16        }
17
18        public override void Work()
19        {
20            int a = aIP.GetCurrentValue(0);
21            int b = bIP.GetCurrentValue(0);
22            resultOP.UpdateToken(a + b);
23        }
24    }
25 }
```

Listing 4.5: Erzeugung von Anschlüssen durch Annotation

```
1  using DynamicNode.Core;
2
3  namespace de.mastersign.demo
4  {
5      public class NonPolymorphB : Node
6      {
7          [InPortInfo("a")]
8          public int A { get; set; }
9
10         [InPortInfo("b")]
11         public int B { get; set; }
12
13         [OutPortInfo("result", typeof(int))]
14         public OutPortHandler ResultHandler;
15
16         public override void Work()
17         {
18             ResultHandler(new Token(A + B));
19         }
20     }
21 }
```

Der Hauptvorteil der Erzeugung von Anschlüssen durch Annotation ist der kompaktere Quell-Code. Der Knotenentwickler muss sich nicht mit den Klassen `InPort` und `OutPort` beschäftigen, sondern kann die Operation des Knotens direkt unter Nutzung der Felder oder Eigenschaften programmieren, welche die Anschlüsse repräsentieren (vgl. *Listing 4.5*). Dieser Vorteil fällt bei den Brückenknoten nicht sehr ins Gewicht, da nur eine kleine Anzahl an Brückenknoten implementiert werden muss. Ein Nachteil ist, dass der Knotenentwickler einen Großteil der Kontrolle abgibt, die ihm das direkte Verwenden der Klassen `InPort` und `OutPort` bereitstellen würde. Wo es dem Entwickler mit der Klasse `InPort` z. B. möglich ist, mehrere Typen für die Kompatibilität des Eingangs mit Ausgängen anzugeben oder Abhängigkeiten zwischen den Typen von Ausgängen und Eingängen zu definieren, kann bei der Nutzung von Annotationen für jeden Anschluss nur genau ein Typ für die Kompatibilität angegeben werden.

Was die Erzeugung von Anschlüssen durch Annotationen für polymorphe Knoten aber gänzlich untauglich macht, ist die Tatsache, dass mit Annotationen erzeugte Anschlüsse nur im Quelltext hart kodiert werden können. Ein nachträgliches Ändern von Anschlüssen oder das Entfernen von Anschlüssen ist mit dieser Technik nicht möglich. In polymorphen Knotentypen für DynamicNodes müssen Anschlüsse aber zur Laufzeit erzeugt und entfernt oder zumindest deren Kompatibilität geändert werden. Eine deklarative Erzeugung durch Annotationen ist folglich nicht möglich.

Zur Erzeugung und Konfiguration von Anschlüssen werden die Klassen `InPort` und `OutPort` verwendet (vgl. *Listing 4.4*).

Initialisierungsphase

Zur Durchführung der polymorphen Initialisierungsphase muss die Adresse einer Interaktion durch den Datenflussprogrammierer ausgewählt worden sein. Ein Brückenknoten sollte dem Programmierer eine Benutzeroberfläche in Form einer *Knotensteuerung* anbieten (vgl. *Benutzeroberfläche* auf Seite 55), welche die Auswahl einer Interaktion, in Form von Adressen für Klassen und Klassenmitgliedern, ermöglicht. Während der Lebenszeit eines Brückenknotens, sollte es über die Knotensteuerung jederzeit möglich sein, ihn an eine neue Interaktion zu binden.

Die Hauptaufgabe der Initialisierungsphase von polymorphen Knoten ist die Erzeugung, Konfiguration und evtl. die Zerstörung von Anschlüssen an einem Knoten. In Abhängigkeit der Interaktion gestaltet sich die Konfiguration unterschiedlich komplex. Es gibt Interaktionsformen, die unabhängig von der konkreten Interaktion, immer die gleiche Anzahl von Ein- und Ausgängen an einem Brückenknoten benötigen und es gibt Interaktionsformen bei denen die benötigte Anzahl der Ein- und Ausgänge an dem Brückenknoten von der konkret zu bindenden Interaktion abhängt.

Es gibt drei Interaktionsformen, die eine variable Anzahl von Ein- und Ausgängen benötigen. Dabei handelt es sich um den Konstruktor, den statischen und den objektgebundenen Methodenaufruf. Diese drei Interaktionsformen umfassen Interaktionen mit unterschiedlicher Parameteranzahl. Bei statischen und objektgebundenen Methodenaufrufen kommt dazu, dass manche Methoden einen Rückgabewert besitzen, der einen entsprechenden Ausgang erforderlich macht, und andere nicht. Nur diese drei Interaktionsformen benötigen voll polymorphe Knotentypen (vgl. *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen*). Für die übrigen Interaktionsformen genügen schwach polymorphe Knotentypen.

Es bilden sich die beiden folgenden Gruppen von Interaktionsformen:

Schwach polymorph

- Feld lesen
- Feld schreiben
- Eigenschaft lesen
- Eigenschaft schreiben
- Ereignis überwachen
- Statisches Feld lesen
- Statisches Feld schreiben
- Statische Eigenschaft lesen
- Statische Eigenschaft schreiben
- Statisches Ereignis überwachen

Voll polymorph

- Konstruktor aufrufen
- Methode aufrufen
- Statische Methode aufrufen

Konfigurationsalgorithmus für schwach polymorphe Brückenknoten

Der Algorithmus für die erste Gruppe ist einfach und variiert in Abhängigkeit der jeweiligen Interaktionsform. Im folgenden Beispiel wird der Algorithmus für das Schreiben einer objektgebundenen Eigenschaft beschrieben. Ein Brückenknoten für statische Ereignisse benötigt keinen Eingang für einen Objektkontext. Er registriert seinen Ereignis-Handler einmal während seiner Initialisierung und meldet ihn erst vor seiner Zerstörung wieder ab.

Ein Knoten für das Lesen von nicht statischen Eigenschaften benötigt einen Eingang für das Objekt, welches als Kontext verwendet werden soll (*instance*), und einen Eingang für den neuen Wert (*value*). Der Algorithmus ist demzufolge wie folgt aufgebaut:

1. Ermittle die Klasse, in der die Eigenschaft definiert ist
2. Setze die Typenkompatibilität des Eingangs *instance* auf diese Klasse
3. Ermittle den Datentyp der Eigenschaft
4. Setze die Typenkompatibilität des Eingangs *value* auf diesen Datentyp

Da die Adresse der Eigenschaft durch ein Objekt der Klasse `System.Reflection.PropertyInfo`[1] angegeben wird, kann die definierende Klasse mit der Eigenschaft `MemberInfo.DeclaringType`[2] abgefragt werden. Der Datentyp der Eigenschaft kann über die Eigenschaft `PropertyInfo.PropertyType`[3] ermittelt werden.

Um die Typenkompatibilität eines Eingangs schnell anpassen zu können, bietet sich die Hilfsklasse `DynamicNode.Core.InPortSingleDataTypeProvider`[4] an. Ein Objekt dieser Klasse kann bei der Erzeugung eines Eingangs als Kompatibilitätsbeschreibung übergeben werden, da sie die Schnittstelle `DynamicNode.Core.IInPortDataTypeProvider`[5] implementiert. Über die Eigenschaft `InPortSingleDataTypeProvider.SupportedDataType`[6] kann dann jederzeit die Typenkompatibilität des Eingangs geändert werden.

Konfigurationsalgorithmus für voll polymorphe Brückenknoten

Für Interaktionsformen der zweiten Gruppe ist der Algorithmus komplizierter. Der Brückenknoten für einen Konstruktor benötigt immer einen Ausgang für das erzeugte Objekt und ein Brückenknoten für einen nicht statischen Methodenaufruf benötigt immer einen Eingang für den Objektkontext. Im Übrigen sind die Ein- und Ausgänge variabel.

Als Beispiel für den Algorithmus soll der komplizierteste Fall beschrieben werden und zwar die Polymorphie für den nicht statischen Methodenaufruf. Solch ein Brückenknoten benötigt, wie schon erwähnt, einen Eingang für das Objekt, welches als Kontext verwendet werden soll (*instance*). Alle übrigen Ein- und Ausgänge müssen aus den Parametern und dem Rückgabewert der Methode abgeleitet werden. Der Algorithmus besteht aus den folgenden Schritten:

[1] http://msdn.microsoft.com/en-us/library/system.reflection.propertyinfo.aspx
[2] http://msdn.microsoft.com/en-us/library/system.reflection.memberinfo.declaringtype.aspx
[3] http://msdn.microsoft.com/en-us/library/system.reflection.propertyinfo.propertytype.aspx
[4] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_InPortSingleDataTypeProvider.htm
[5] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_IInPortDataTypeProvider.htm
[6] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Core_InPortSingleDataTypeProvider_SupportedDataType.htm

1. Zerstöre alle während der letzten Initialisierungsphase erzeugten Ein- und Ausgänge.
2. Für alle Parameter der Methode:
    a) Ermittle den Datentyp des Parameters
    b) Wenn der Parameter ein Eingabeparameter oder ein Referenzparameter ist:
        i. Erzeuge einen Eingang mit dem Namen *in_<Parametername>*
        ii. Setze seine Typenkompatibilität auf den Datentyp des Parameters
    c) Wenn der Parameter ein Referenzparameter oder ein Ausgabeparameter ist:
        i. Erzeuge einen Ausgang mit dem Namen *out_<Parametername>*
        ii. Setze seine Typenkompatibilität auf den Datentyp des Parameters
3. Ermittle die Klasse, in der die Interaktion definiert ist
4. Setze die Typenkompatibilität des Eingangs *instance* auf die Klasse, in der die Interaktion definiert wird
5. Falls die Methode eine Rückgabewert besitzt:
    a) Ermittle den Rückgabedatentyp der Methode
    b) Erzeuge eine Ausgang mit dem Namen *return*
    c) Setze die Typenkompatibilität auf den Rückgabedatentyp der Methode

Der beschriebene Algorithmus lässt sich dahingehend optimieren, dass der Ausgang *return* nicht jedesmal zerstört und anschließend bei Bedarf neu erzeugt wird, sondern dass er nur zerstört oder erzeugt wird, wenn dies basierend auf der letzten Initialisierungsphase wirklich erforderlich ist. Falls der Ausgang *return* in einer Initialisierungsphase nicht erzeugt werden muss, weil er in einer vorangegangenen Initialisierungsphase bereits erzeugt wurde, muss lediglich die Typenkompatibilität angepasst werden.

Um die Typenkompatibilität eines Ausgangs schnell anpassen zu können, bietet sich die Hilfsklasse `DynamicNode.Core.OutPortDataTypeProvider`[1] an. Ein Objekt dieser Klasse kann bei der Erzeugung eines Ausgangs als Kompatibilitätsbeschreibung übergeben werden, da sie die Schnitstelle `DynamicNode.Core.IOutPortDataTypeProvider`[2] implementiert. Über die Eigenschaft `OutPortDataTypeProvider.SupportedDataType`[3] kann auf einfache Weise die Typenkompatibilität des Ausgangs geändert werden.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_OutPortDataTypeProvider.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_IOutPortDataTypeProvider.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Core_OutPortDataTypeProvider_SupportedDataType.htm`

Ausführungsphase

In *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen* wird beschrieben, dass die Ausführungsphase eines polymorphen Knotens die Operation in Abhängigkeit der aktuellen Anschlüsse und deren Konfiguration durchführt.

Unabhängig von dem Grad der Polymorphie lässt sich die Ausführungsphase der Brückenknoten in die folgenden Schritte teilen:

1. **Eingabedaten sammeln**
   Die Werte aller Eingänge mit `InPort.CurrentToken.Value` ermitteln. Eine Umwandlung des Datenformats (*Marshalling*) ist für die Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken nicht notwendig.
   Am Beispiel des Brückenknotens für das Lesen eines objektgebundenen Feldes muss nur der Objektkontext vom Eingang *instance* gelesen werden.

2. **Interaktion ausführen**
   Die Ausführung der Interaktion ist für jede Interaktionsform unterschiedlich. In diesem Schritt erfolgt die Bindung der Interaktion, also z. B. die Übergabe von Parametern und der Aufruf bei einer Methode.
   Am Beispiel des Brückenknotens für das Lesen eines objektgebundenen Feldes muss in diesem Schritt `Object FieldInfo.GetValue(Object)` aufgerufen werden, wobei der in Schritt 1 gelesenen Objektkontext übergeben und der Rückgabewert zwischengespeichert wird.

3. **Ausgabedaten weitergeben**
   Alle Werte, die in Schritt 2 von der Interaktion als Rückgabewerte erhalten wurden, werden über die überladene Methode `void OutPort.UpdateToken(...)`[1] über den entsprechenden Ausgang weitergegeben.
   Am Beispiel des Brückenknotens für das Lesen eines objektgebundenen Feldes muss der Rückgabewert aus Schritt 2 über den Ausgang *value* weitergegeben werden.

4. **Kontrollmarke weitergeben**
   Wie nachfolgend in *Unterabschnitt 4.3.3 Anschlüsse für die Einbettung in einen Kontrollfluss* erläutert, muss der Brückenknoten zur Einbettung in einen Kontrollfluss (vgl. *Kontrollfluss* auf Seite 54) nach Abschluss der Operation eine Steuermarke über einen dafür vorgesehen Ausgang weitergeben.

Für die Brückenknoten unterscheidet sich die Ausführungsphase ebenso wie die Initialisierungsphase deutlich zwischen voll polymorphen und schwach polymorphen Brückenknoten. Bei schwach polymorphen Brückenknoten sind Schritt 1 und 3 ein fester Algorithmus, da

[1] `http://dynamicnodes.mastersign.de/docs/api/html/Overload_DynamicNode_Core_OutPort_UpdateToken.htm`

durch die Interaktionsform bereits alle Informationen für das Sammeln und Weitergeben der Daten vorhanden sind. Bei voll polymorphen Bückenknoten sind alle Schritte von der gebundenen Interaktion abhängig.

So hängt die Menge der Eingänge, deren aktuelle Markenwerte in Schritt 1 gelesen werden müssen, immer von der aktuell gebundenen Interaktion ab. Ebenso ist die Übergabe der Eingabewerte an die Interaktion in Schritt 2 von den Parametern der Interaktion abhängig und davon, ob sie statisch ist oder nicht. Und auch die Weitergabe der Ergebniswerte von der Interaktion an die Ausgänge ist davon abhängig, ob es Ausgabeparameter und einen Rückgabewert gibt.

Für voll polymorphe Knoten konzentriert sich Schritt 2 auf den Aufruf von `Object MethodBase.Invoke(Object, Object[])`. Dazu wird in Schritt 1 ein `Object`-Array mit allen Eingabeparameterwerten gebildet, welches an `Invoke()` übergeben wird. Hat die gebundene Interaktion keinen Objektkontext, wird als erster Parameter an dessen Stelle `null` übergeben. Hat die Interaktion keinen Rückgabewert, gibt `Invoke()` `null` zurück.

4.3.5 Benutzerschnittstelle zur Konfiguration der Polymorphie

In diesem Unterabschnitt wird die Benutzerschnittstelle zur Konfiguration der Polymorphie entworfen. Der da die Brückenknoten für DynamicNodes polymorphe Knoten sind und ihre Initialisierungsphase somit nach der Instanzierung des Brückenknotens beliebig auft aufgerufen werden kann (vgl. *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen*), wird einen Benutzerschnittstelle benötigt, welche die Parametrisierung der Initialisierungsphase und die Ausführung der Initialisierungsphase erlaubt.

Parametrisierung bedeutet für die Brückenknoten die Adressierung einer Interaktion in den .NET-Klassenbibliotheken. Jede Interaktion wird beschrieben durch eine Interaktionsform und das Klassenmitglied auf das sich die Interaktion beziehen soll. Die Interaktionsform wird durch den Brückenknoten selbst vorgegeben. Die Adresse des Klassenmitglieds muss über die Benutzerschnittstelle des Brückenknoten auswählbar sein.

Die Konfiguration eines instanzierten Knotens ist in DynamicNodes über die Ansicht „Knotensteuerung" möglich, sofern der Knotentyp ein Benutzersteuerelement als Knotensteuerung zur Verfügung stellt (vgl. 4.1.1). Sollen die Brückenknoten also eine Benutzerschnittstelle zur Konfiguration anbieten, müssen sie ein Benutzersteuerelement dafür anbieten. Dieses Steuerelement dient im Falle der Brückenknoten lediglich der Adressauswahl eines Klassenmitglieds und wird deshalb nachfolgend Adressauswahlsteuerelement genannt.

Die Adressen von Klassenmitgliedern sind immer realitv zu der Klasse, für die das Mitglied definiert wurde. Demzufolge besteht jedes Adressauswahlsteuerelement für einen Brückenknoten aus zwei Teilen: Einer Adressauswahl für eine Klasse – dieser Teil ist für alle Brücken-

knoten gleich – und eine Adressauswahl für eine bestimmte Form von Klassenmitglied. Für jede Form von Klassenmitglied muss ein eigenes Adressauswahlsteuerelement zur Verfügung gestellt werden.

Da sich Benutzersteuerelemente verschachteln lassen, bietet es sich an, ein Benutzersteuerelement für die Auswahl der Adresse einer Klasse zu implementieren und dieses Steuerelement in den verschiedenen Adressauswahlsteuerelementen der Brückenknoten wiederzuverwenden.

4.3.6 Visuelle Gestaltung der Brückenknoten

In diesem Unterabschnitt wird die Visualisierung der Brückenknoten entworfen. Dazu muss entschieden werden, ob und wie der Knoten eine eigene Visualisierungsfläche zeichnet, ob der Name des Knotentyps ausgeblendet und wie die Anschlüsse angeordnet werden sollen (vgl. *Benutzeroberfläche* auf Seite 55).

In *Unterabschnitt 4.3.1 Auswahl des Lösungsansatzes* wurde entschieden, dass die Brückenknoten durch polymorphe Brückenknoten je Interaktionsform umgesetzt werden sollen. Dadurch entsteht die Möglichkeit, dass es in einem Graphen mehrere Instanzen des gleichen Brückenknotentyps gibt. Z. B. könnte ein Graph mehrere Brückenknoten für Methodenaufrufe enthalten. Damit der Programmierer die verschiedenen Methodenaufrufe unterscheiden kann, ist eine unterschiedliche Visualisierung der einzelnen Brückenknoten notwendig. Die Standardvisualisierung, die DynamicNodes für die Knoten anbietet, stellt jedoch alle Instanzen desselben Brückenknotentyps gleich dar.

Um die verschiedenen, durch Brückenknoten gebundenen Interaktionen, im Editor von DynamicNodes unterscheiden zu können, sollte die Visualisierung die Adresse der Interaktion, in einer für den Menschen leicht verständlichen Form, enthalten. Für diesen Zweck wird der Name der Klasse, in dem die Interaktion definiert ist, und der Name der Interaktion in Textform dargestellt. Zusätzlich wird der Namensraum der Klasse mit einer kleinen Schriftart und einer farblich abgeschwächten Füllung in Textform ausgegeben.

Damit die verschiedenen Interaktionsformen leicht unterschieden werden können, sollte die Visualisierung auch ein passendes Symbol für jede Interaktionsform darstellen. Da die Programmierung mit Brückenknoten für .NET-Klassenbibliotheken allgemeines Wissen über die .NET-Programmierung voraussetzt und das Hauptwerkzeug für die .NET-Programmierung die Entwicklungsumgebung Microsoft Visual Studio ist, ist es sinnvoll die Symbolsprache von Visual Studio zu übernehmen. Alle notwendigen Symbole finden sich in der Grafik-Bibliothek, die Microsoft mit Visual Studio ausliefert. Ein kleiner Ausschnitt der Grafikbibliothek ist in *Abbildung 4.5* dargestellt. Einige der Symbole wurden speziell für die Brückenknoten angepasst. Für Schreib- und Leseinteraktionen, wie z. B. „Feld lesen" oder „statische

Eigenschaft schreiben", sollte das Symbol für Feld oder Eigenschaft um ein Symbol für Lesen oder Schreiben ergänzt werden.

Abbildung 4.5: Symbole für Typen und Interaktionsformen

Da die Interaktionsform der gebundenen Interaktion durch ein grafisches Symbol visualisiert wird, ist die Ausgabe des Namens des Knotentyps, entsprechend der Standardvisualisierung, nicht mehr notwendig und kann deaktiviert werden.

Mit diesen Vorgaben ergibt sich ein zweizeiliges Layout für die Gestaltung des Rechtecks, welches den Knoten visualisiert (vgl. *Abbildung 4.6*).

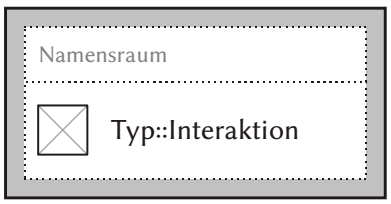


Abbildung 4.6: Layout der grafischen Darstellung eines Brückenknotens

Die Anschlüsse sollten nach einem einheitlichen Schema an allen Brückenknoten angeordnet werden. Dabei gelten die folgenden Regeln:

1. Anschlüsse für die Einbettung in einen Kontrollfluss werden an der linken Kante angeordnet – oben Eingänge, unten Ausgänge.

2. Der Eingang für eine Instanz als Kontext für nicht statische Interaktionen wird an der Oberkante ganz links angeordnet.
3. Eingänge für Parameter werden an der oberen Kante rechts angeordnet.
4. Der Ausgang für einen Rückgabewert oder die Instanz bei einem Konstruktor wird an der unteren Kante ganz links angeordnet.
5. Ausgänge von Ausgabe- oder Referenzparametern werden an der Unterkante rechts angeordnet.
6. Ausgänge für die Daten eines Ereignisses werden an der Unterkante angeordnet.

4.3.7 Klassenhierarchie der Brückenknoten

Die Brückenknoten werden als .NET-Klassen implementiert. Das DynamicNodes-Programmiersystem fordert von einem Knoten, dass er die Schnittstelle `DynamicNode.Core.INode`[1] (vgl. *Abbildung A.3* im Anhang auf Seite 151) implementiert. Da diese Schnittstelle recht umfangreich ist, und ein großer Anteil der durch sie geforderten Funktionalität für die meisten Knoten gleich ist, bietet die API von DynamicNodes die abstrakte Basisklasse `DynamicNode.Core.Node`[2] (vgl. *Abbildung A.4* im Anhang auf Seite 152), welche diese Schnittstelle nahezu vollständig implementiert.

Aus Gründen, die an dieser Stelle nicht näher erläutert werden sollen, stellt DynamicNodes mit `DynamicNode.Core.StreamNode`[3] (vgl. *Abbildung A.5* im Anhang auf Seite 152) eine weitere abstrakte Klasse als Spezialisierung von `Node` zur Verfügung. Beide Klassen bieten keine Unterstützung, um die visuelle Darstellung der Knoten zu kontrollieren. Aus diesem Grund gibt es eine weitere Spezialiserung in der Klasse `DynamicNode.Ext.Visual.VisualNode`[4] (vgl. *Abbildung A.6* im Anhang auf Seite 153). Mit `VisualNode` als Basisklasse ist die Implementierung von Knotentypen am einfachsten, deshalb sollen die Brückenknoten von dieser Klasse abgeleitet werden.

Um die visuelle Darstellung der Knoten einheitlich nach *Unterabschnitt 4.3.6 Visuelle Gestaltung der Brückenknoten* zu gestalten, soll eine gemeinsame Basisklasse für alle Brückenknoten verwendet werden, welche von `VisualNode` erbt. Diese Klasse heißt `AbstractWrapper`. Der Begriff *Wrapper* (Kapsel) wird gewählt, weil alle Typen in DynamicNodes englische Namen besitzen und der Vorgang des Einbindens einer Interaktion als *wrappen* übersetzt wird. Das Präfix *Abstract* wird vorangestellt, um die Funktion als Basisklasse bereits im Namen ersichtlich zu machen.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_INode.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_Node.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Core_StreamNode.htm`
[4] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Visual_VisualNode.htm`

Alle Interaktionen, welche eine Parameterliste benötigen (Konstruktor, objektgebundene Methode, statische Methode), erhalten eine gemeinsame Basisklasse, in welcher die Polymorphie implementiert wird. Diese Basisklasse heißt `AbstractMethodBase`. Der Name *MethodBase* wird deshalb gewählt, weil die Metadatenklassen `MethodInfo`[1] und `ConstructorInfo`[2], welche die Adressen für die zu bindenden Interaktionen bilden, ebenfalls eine gemeinsame Basisklasse `System.Reflection.MethodBase`[3] besitzen. Diese Basisklasse dient `AbstractMethodBase` als abstrakte Adresse für die einheitliche Behandlung von Konstruktoren und Methoden. Die Implementierung der Polymorphie für die Konfiguration der Ein- und Ausgänge, welche die Parameter der Interaktion vertreten, kann sich dabei auf die Methode `MethodBase.GetParameters()`[4] stützen.

Die Klasse `MethodBase` besitzt jedoch keine Eigenschaften oder Methoden um einen Rückgabewert zu ermitteln. Dies ist erst mit den abgeleiteten Klassen `MethodInfo` durch `MethodInfo.ReturnType`[5] und mit `ConstructorInfo`[6] durch `ConstructorInfo.DeclaringType` möglich. Deshalb deklariert `AbstractMethodBase` zwei abstrakte Methoden: `Type GetReturnType()` und `string GetReturnOutPortId()`, welche durch den Brückenknotentypen für Konstruktoren und die Brückenknotentypen für Methoden implementiert werden müssen.

Auch für die beiden Interaktionen „objektgebundene Methode" und „statische Methode" lassen sich einige Aufgaben in einer gemeinsamen Basisklasse `AbstractMethod` implementieren. Dazu gehört die Implementierung der beiden abstrakten Methoden `Type GetReturnType()` und `string GetReturnOutPortId()`. Von `AbstractMethod` werden die beiden konkreten Brückenknotentypen `NMethod` und `NStaticMethod` abgeleitet. Laut den Benennungskonventionen von DynamicNodes beginnen die Namen von konkreten Klassen, die Knotentypen implementieren, immer mit einem großen „N".

Die Vererbungshierarchie der Brückenknotentypen für Methoden und Konstruktoren ist in *Abbildung 4.7* als Unified Modeling Language (UML)-Diagramm dargestellt.

[1] http://msdn.microsoft.com/de-de/library/system.reflection.methodinfo.aspx
[2] http://msdn.microsoft.com/de-de/library/system.reflection.constructorinfo.aspx
[3] http://msdn.microsoft.com/de-de/library/system.reflection.methodbase.aspx
[4] http://msdn.microsoft.com/de-de/library/system.reflection.methodbase.getparameters.aspx
[5] http://msdn.microsoft.com/de-de/library/system.reflection.methodinfo.returntype.aspx
[6] http://msdn.microsoft.com/de-de/library/system.reflection.memberinfo.declaringtype.aspx

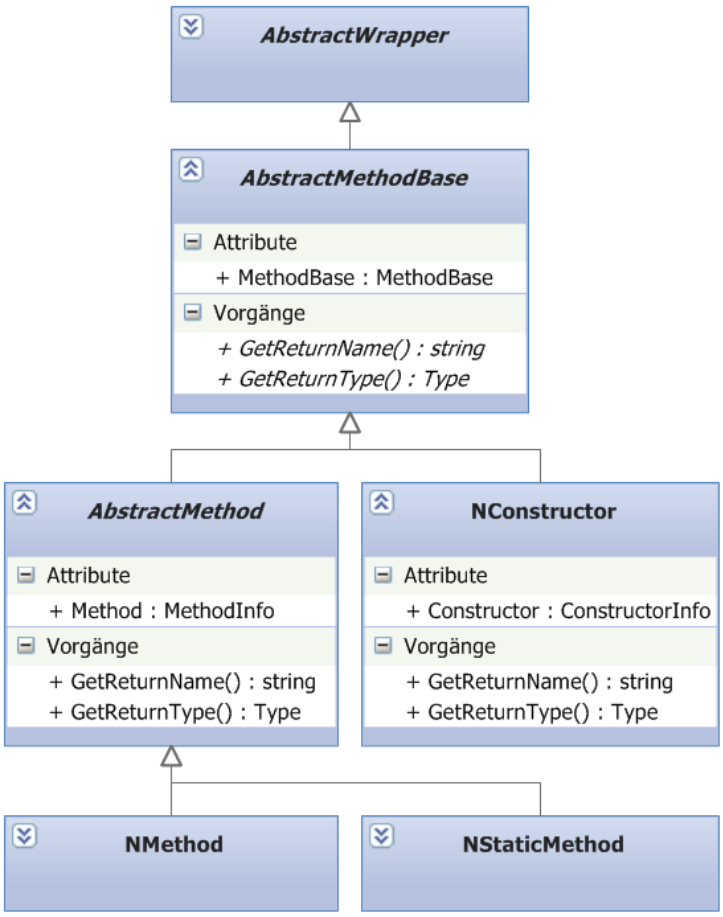


Abbildung 4.7: Vererbungshierarchie von `NMethod`, `NStaticMethod` und `NConstructor`

Wird die Klassenhierarchie der Metadatenklassen, welche zur Adressierung verwendet werden (`MethodBase` und Kindklassen), mit der Klassenhierarchie der Brückenknotenklassen (`AbstractWrapper` und Kindklassen) verglichen, wird deutlich, dass sie einem gemeinsamen Muster folgen. *Abbildung 4.8* stellt beide Hierarchien einander gegenüber.

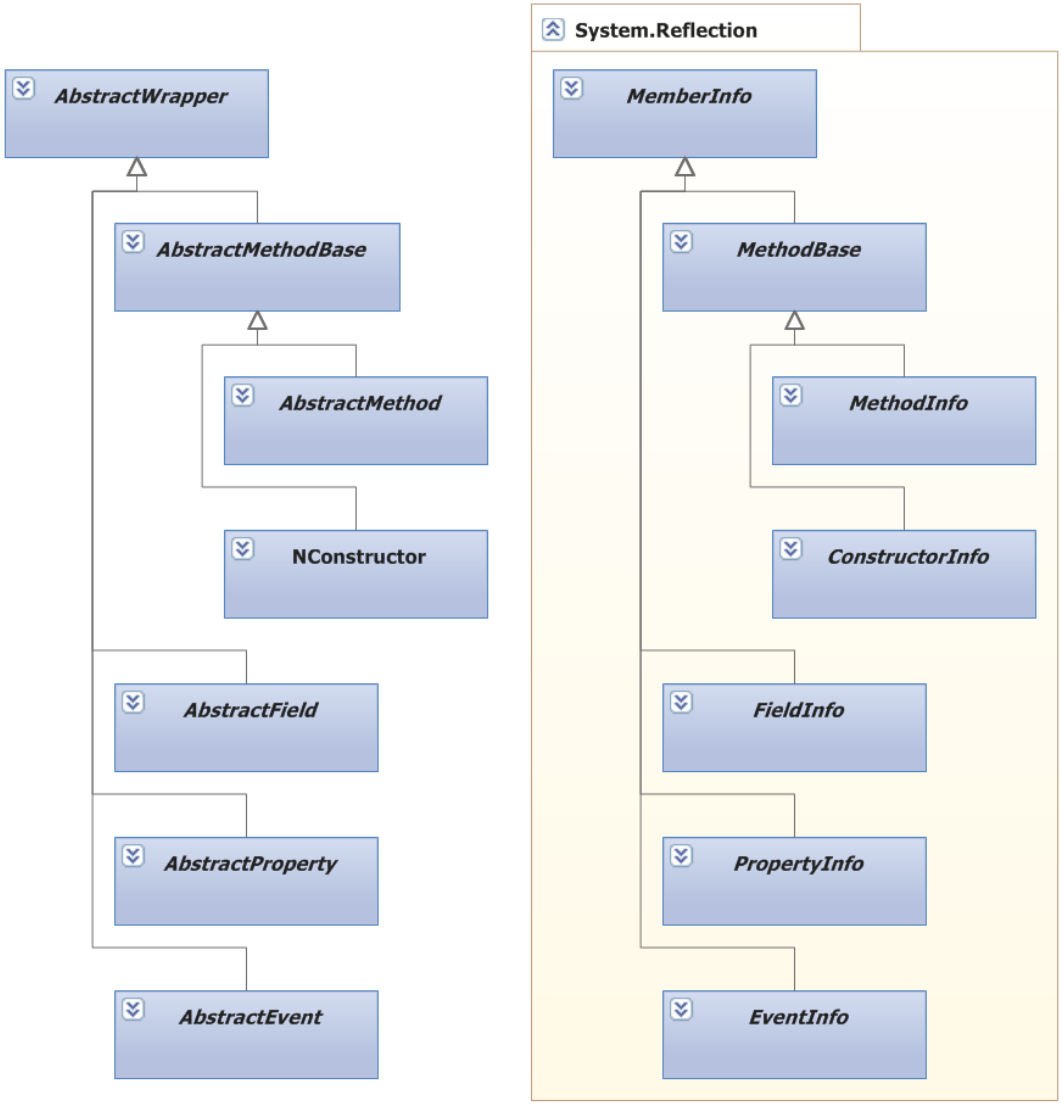


Abbildung 4.8: Vergleich: Basisklassen der Brückenknoten und Adressklassen

Zu jeder Metadatenklasse aus dem Namensraum `System.Reflection` gibt es eine Basisklasse für Brückenknoten, welche die Adresse der Interaktion in Form eines Metadatenobjektes als Eigenschaft zur Verfügung stellt. So besitzt die Klasse `AbstractMethodBase` eine Eigenschaft `MethodBase` vom Typ `System.Reflection.MethodBase` und die Klasse `AbstractMethod` eine Eigenschaft `Method` vom Typ `System.Reflection.MethodInfo`. Eine Ausnahme bildet die Klasse `NConstructor`, welche bereits eine konkrete Brückenknotenklasse ist. Denn für einen Konstruktor gibt es nur eine Interaktionsform – den Aufruf.

Diese Parallele verfolgend, entstehen drei weitere Basisklassen: `AbstractField`, `AbstractProperty` und `AbstractEvent`. Jede von ihnen stellt die Adresse der zu bindenden Interaktion als Eigenschaft zur Verfügung (vgl. *Abbildung 4.9*, *Abbildung 4.10* und *Abbildung 4.11*).

Die Basisklasse `AbstractField` bietet die Eigenschaft `Field` vom Typ `FieldInfo` an. Von ihr

werden die vier konkreten Brückenknotenklassen `NRead`, `NWrite`, `NStaticRead` und `NStaticWrite` abgeleitet (vgl. *Abbildung 4.9*). Die Klasse `NRead` ist der Brückenknotentyp für das Lesen von objektgebundenen Feldern. Die Klasse `NWrite` ist der Brückenknotentyp für das Schreiben von objektgebundenen Feldern. Die Klasse `NStaticRead` ist der Brückenknotentyp für das Lesen von statischen Feldern. Und die Klasse `NStaticWrite` ist der Brückenknotentyp für das Schreiben von statischen Feldern.

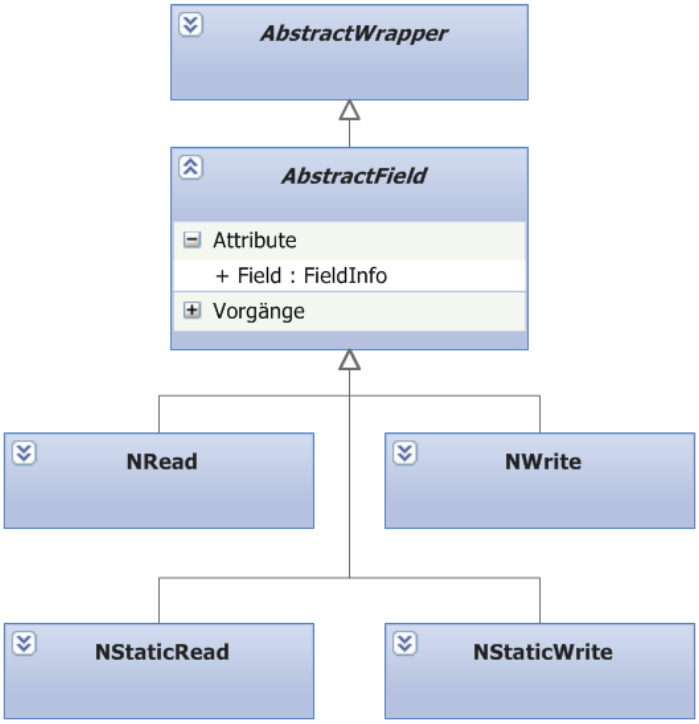


Abbildung 4.9: Vererbungshierarchie von `NRead`, `NStaticRead`, `NWrite` und `NStaticWrite`

Die Basisklasse `AbstractProperty` bietet die Eigenschaft `Property` vom Typ `PropertyInfo` an. Von ihr werden die vier konkreten Brückenknotenklassen `NGet`, `NSet`, `NStaticGet` und `NStaticSet` abgeleitet (vgl. *Abbildung 4.10*). Die Klasse `NGet` ist der Brückenknotentyp für das Lesen von objektgebundenen Eigenschaften. Die Klasse `NSet` ist der Brückenknotentyp für das Schreiben von objektgebundenen Eigenschaften. Die Klasse `NStaticGet` ist der Brückenknotentyp für das Lesen von statischen Eigenschaften. Die Klasse `NStaticSet` ist der Brückenknotentyp für das Schreiben von statischen Eigenschaften.

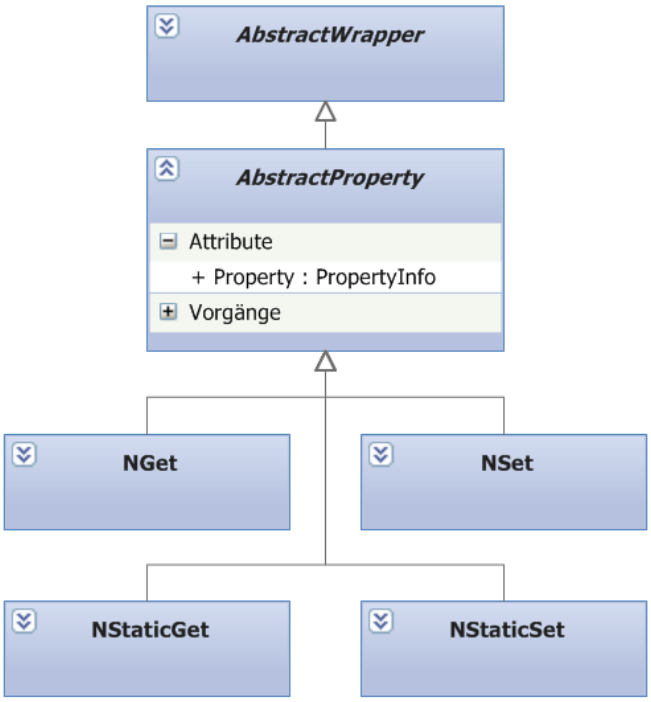


Abbildung 4.10: Vererbungshierarchie von `NGet`, `NStaticGet`, `NSet` und `NStaticSet`

Die Basisklasse `AbstractEvent` bietet die Eigenschaft `Event` vom Typ `EventInfo` an. Von ihr werden die beiden konkreten Brückenknotenklassen `NEvent` und `NStaticEvent` abgeleitet (vgl. *Abbildung 4.11*). Die Klasse `NEvent` ist der Brückenknotentyp für das Überwachen von objektgebundenen Ereignissen. Die Klasse `NStaticEvent` ist der Brückenknotentyp für das Überwachen von statischen Ereignissen.

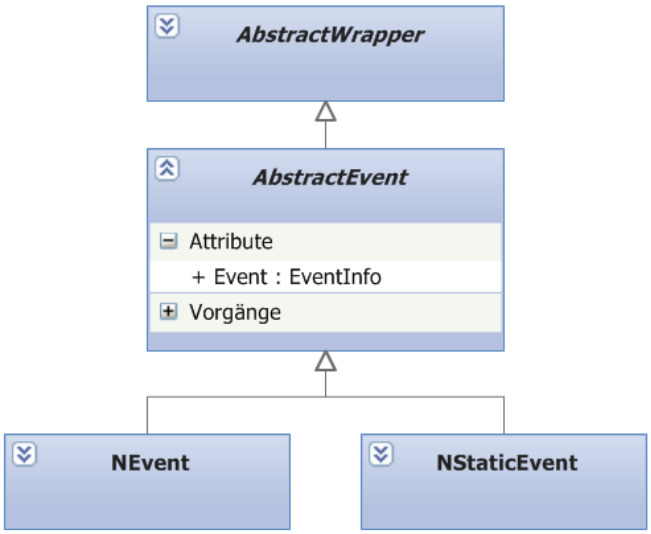


Abbildung 4.11: Vererbungshierarchie von `NEvent` und `NStaticEvent`

Die Basisklassen bieten nicht nur die jeweilge Adresse für die Interaktion als Eigenschaft

an, sie implementieren auch eine Benutzerschnittstelle zur Auswahl der Adresse. Dazu überschreiben sie die Methode `VisualNode.InitControlPanel(NodeControlPanelExtension ctrlPanel)`[1] und richten ein Steuerelement zur Auswahl von Klasse und Mitglied als Knotensteuerung ein. Da sich die Auswahlmöglichkeiten nach der Metadatenklasse der Adresse richten, ist für jede Basisklasse der Brückenknoten ein anderes Steuerelement zur Auswahl der Interaktionsadresse erforderlich. Damit nicht zu viele verschiedene Adressauswahlsteuerelemente notwendig werden, sollten diese derart implementiert werden, dass sie entweder für die Auswahl von objektgebundenen oder für die Auswahl von statischen Klassenmitgliedern geeignet sind.

Es werden die folgenden Adressauswahlsteuerelemente benötigt: `MethodSelector`[2], `ConstructorSelector`[3], `FieldSelector`[4], `PropertySelector`[5] und `EventSelector`[6]. Da alle diese Steuerelemente die Auswahl einer Klasse bzw. eines Typs ermöglichen müssen, bietet es sich an, dafür ein dediziertes Steuerelement `TypeSelector`[7] zu implementieren.

Die vollständige Klassenhierarchie aller Brückenknoten und ihrer Basisklassen bis zurück zur Klasse `VisualNode` ist in *Abbildung 4.12* dargestellt.

[1] http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Visual_VisualNode_InitControlPanel.htm
[2] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_MethodSelector.htm
[3] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_ConstructorSelector.htm
[4] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_FieldSelector.htm
[5] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_PropertySelector.htm
[6] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_EventSelector.htm
[7] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_TypeSelector.htm

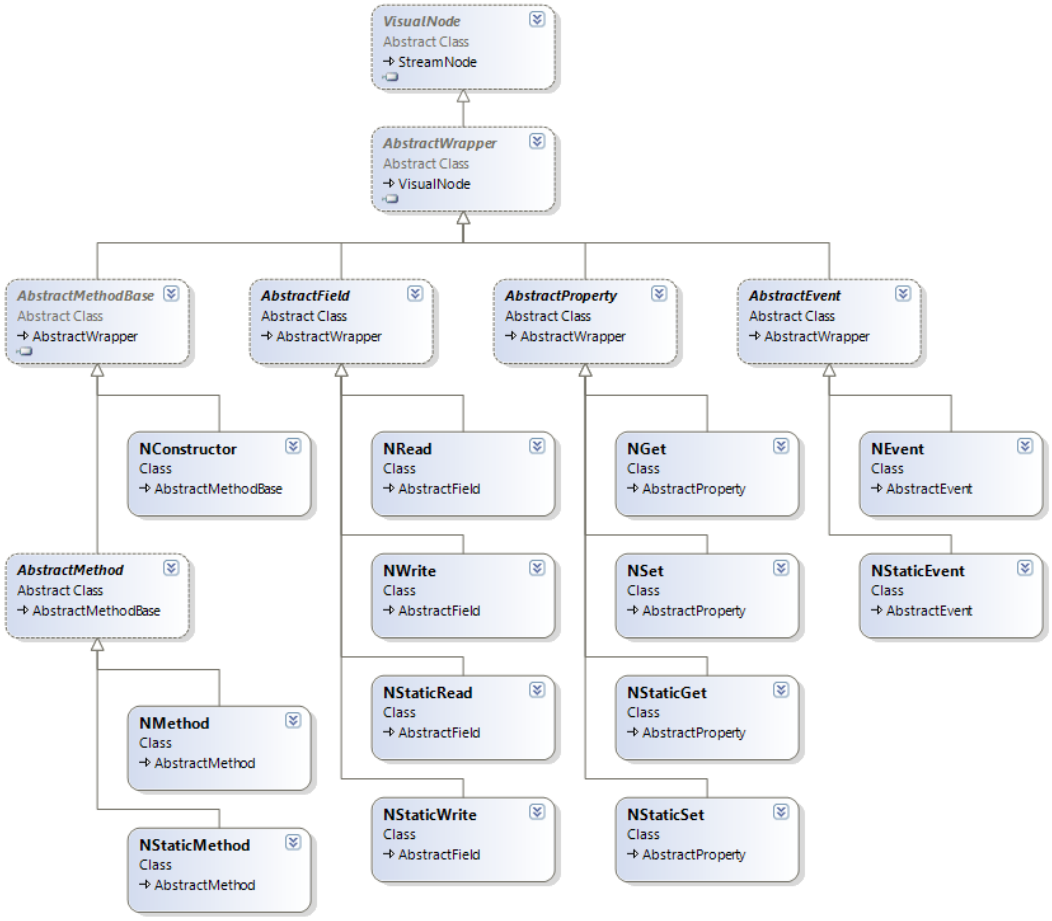
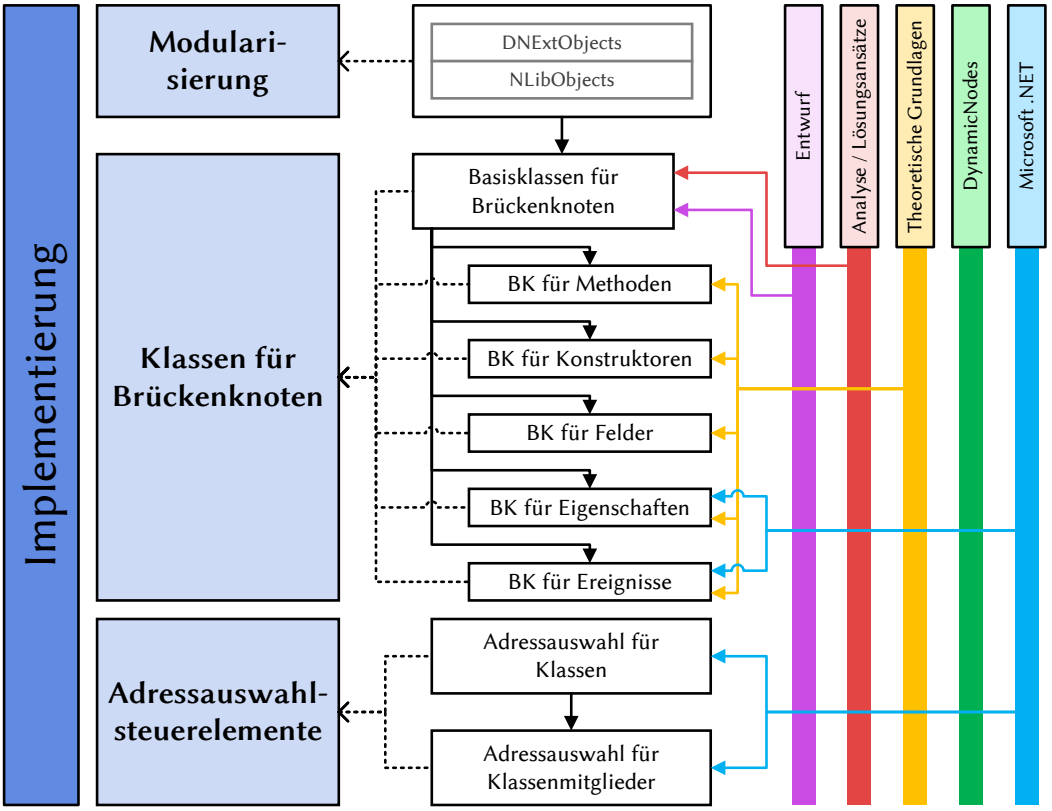


Abbildung 4.12: Klassenhierarchie der Brückenknoten

5 Implementierung



BK = Brückenknoten

Dieses Kapitel führt den praktischen Anteil der Arbeit fort und beschreibt die Implementierung der in *Kapitel 4 Entwurf* entwickelten Brückenknoten. Das Kapitel gliedert sich in drei Abschnitte.

In Abschnitt 5.1 wird erklärt wie die Implementierung in zwei Assemblies aufgeteilt wird. Anschließend wird in Abschnitt 5.2 die Implementierung der einzelnen Klassen aus *Unterabschnitt 4.3.7 Klassenhierarchie der Brückenknoten* erläutert. Im letzten Abschnitt dieses Kapitels – Abschnitt 5.3 – werden die Benutzeroberflächensteuerelemente beschrieben, welche die Schnittstelle für den Datenflussprogrammierer darstellen, um die polymorphen Brückenknoten zu konfigurieren.

5.1 Modularisierung

Die Klassen zur Implementierung der Brückenknoten sind in zwei Assemblies aufgeteilt. Ein Assembly enthält alle konkreten Brückenknotentypen und die interaktionsspezifischen Basisklassen (`AbstractMethod`, `AbstractField`, `AbstractProperty` und `AbstractEvent`). Dieses Assembly bildet die eigentliche Knotenbibliothek für DynamnicNodes. Die Namen von Knotenbibliotheken für DynamicNodes besitzen üblicherweise den Präfix „NLib". Die Knotenbibliothek der Brückenknoten heißt *NLibObjects*. Der Namensraum in dem sich alle Klassen dieses Assemblies befinden ist `DynamicNode.Lib.Objects`.

Das zweite Assembly enthält die Basisklassen `AbstractWrapper` und `AbstractMethodBase`. Diese beiden Klassen werden in ein separates Assembly ausgegliedert, weil sie mit ihrer Schnittstelle, und vor allem mit der Implementierung für Polymorphie (`AbstractMethodBase`), auch für Brückenknoten zu anderen Technologien verwendet werden können. Des Weiteren befinden sich die Steuerelemente für die Adressauswahl in dieser zweiten Assembly. Auch eine Klasse mit Hilfsfunktionen für den Umgang mit Metadaten (`CLRTools`) befindet sich in diesem Assembly.

Da dieses Assembly auch Klassen enthält, welche zur Erweiterung der Benutzeroberfläche von DynamicNodes verwendet werden können, wird dieses Assembly als Erweiterung für DynamicNodes betrachtet und erhält den dafür passenden Namen *DNExtObjects*. Alle Klasse in *DNExtObjects* befinden sich in dem Namensraum `DynamicNode.Ext.Objects`.

Die Assembly *NLibObjects* ist von der Assembly *DNExtObjects* abhängig (vgl. *Abbildung 5.1*). Beide sind von den Assemblies *DNCore*[1] und *DNVisual*[2] abhängig[3]. Die ausführbare Assembly *DNWinApp*[4] ist von *DNExtObjects* abhängig, da sie Klassen aus *DNExtObjects* zur Erweiterung der Benutzeroberfläche verwendet.

[1] `http://dynamicnodes.mastersign.de/architektur/DNCore.php`
[2] `http://dynamicnodes.mastersign.de/architektur/DNVisual.php`
[3] `http://dynamicnodes.mastersign.de/architektur/`
[4] `http://dynamicnodes.mastersign.de/architektur/DNWinApp.php`

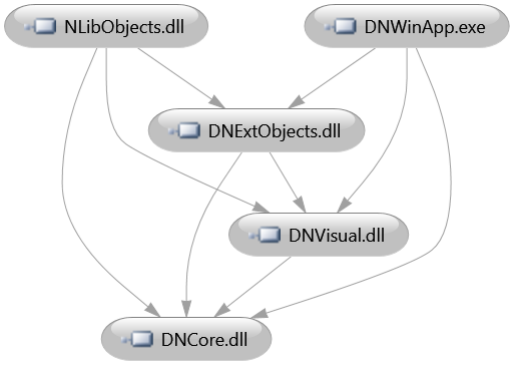


Abbildung 5.1: Abhängigkeiten zwischen einigen Assemblies des DynamicNodes-Systems

5.2 Klassen für Brückenknoten

In diesem Abschnitt werden die einzelnen Klassen der Brückenknoten erläutert.

5.2.1 Basisklassen für Brückenknoten

Zwei Basisklassen in der Klassenhierarchie der Brückenknoten sind unabhängig von den Interaktionsformen. Zum einen die abstrakte Klasse `DynamicNode.Ext.Objects.AbstractWrapper`, welche die Basisklasse für alle Brückenknoten ist, und zum anderen die abstrakte Klasse `DynamicNode.Ext.Objects.AbstractMethodBase`, welche die Basisklasse für die Brückenknoten für Methodenaufrufe (`NMethod`, `NStaticMethod`) und den Brückenknoten für Konstruktoraufrufe (`NConstructor`) ist.

AbstractWrapper

Die Klasse `DynamicNode.Ext.Objects.AbstractWrapper`[1] (vgl. *Abbildung A.7* im Anhang auf Seite 153 und *Listing A.10* im Anhang auf Seite 162) ist die Basisklasse für alle Brückenknoten. Diese Klasse befindet sich in dem Assembly `DNExtObjects` (vgl. *Abbildung 5.1*). Wie in *Unterabschnitt 4.3.7 Klassenhierarchie der Brückenknoten* beschrieben, dient diese Klasse hauptsächlich der Implementierung der Visualisierung des Knotens. Für diesen Zweck überschreibt sie die Methode `VisualNode.PaintCustomArea(Graphics, RectangleF, IPaintProperties)`[2]. In dieser Methode wird der knotenspezifische Anteil der Visualisierung gezeichnet.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_AbstractWrapper.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Visual_VisualNode_PaintCustomArea.htm`

Für das Zeichnen des in *Unterabschnitt 4.3.6 Visuelle Gestaltung der Brückenknoten* entworfenen zweizeiligen Layouts werden die Zeichenketten für zwei Textzeilen und die Grafiken für ein bzw. zwei Symbole benötigt. Da diese Informationen für jede Interaktionsform unterschiedlich sind und die Basisklasse `AbstractWrapper` sie folglich nicht kennen kann, definiert sie vier abstrakte bzw. virtuelle Methoden, die in abgeleiteten Klassen implementiert werden müssen bzw. überschrieben werden können. Durch einen Aufruf dieser Methoden ermittelt `AbstractWrapper` alle für das Zeichnen benötigten Informationen. Die vier Methoden haben die folgenden Signaturen:

- `abstract string GetTextLine1()`
  Diese Methode wird aufgerufen, um die erste Textzeile für die knotenspezifische Visualisierung zu ermitteln.
- `abstract string GetTextLine2()`
  Diese Methode wird aufgerufen, um die zweite Textzeile für die knotenspezifische Visualisieriung zu ermitteln.
- `virtual Image GetInteractionSymbol1()`
  Diese Methode wird aufgerufen, um das Hauptsymbol für die Interaktionsform zu ermitteln. Die Standardimplementierung gibt das Symbol für eine Klasse zurück.
- `virtual Image GetInteractionSymbol2()`
  Diese Methode wird aufgerufen, um das optionale Zusatzsymbol für die Interaktion zu ermitteln. Die Standardimplementierung dieser Methode gibt `null` zurück. Diese Methode braucht nur von Brückenknoten überschrieben zu werden, welche den Zugriff auf ein Feld oder eine Eigenschaft anbinden. Die Implementierung in diesen Brückenknoten soll ein Symbol zurückgegeben, welches anzeigt ob das gebundene Feld bzw. die gebundene Eigenschaft gelesen oder geschrieben wird.

Die Methode `void UpdateVisualization()` ist als geschützt (`protected`) markiert, was bedeutet, dass sie in Kindklassen aufgerufen werden darf, nicht jedoch von Klassen außerhalb der Vererbungshierarchie. Diese Methode führt die folgenden Aufgaben durch: Neuberechnung der erforderlichen Breite der Knotenvisualisierung, Anordnen der Anschlüsse und Neuzeichnen der Knotenvisualisierung. Dazu delegiert sie die Aufgaben an die folgenden Methoden: `GetMaxClientWidth()`, `ConfigPortVisualisation()`, `NodeVisualization.ArrangePorts()`[1] und `NodeVisualization.RepaintNode()`[2].

`AbstractWrapper` deklariert die abstrakte Methode `void InitPorts()`[3], welche in abgelei-

[1] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Visual_NodeVisualization_ArrangePorts.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Visual_NodeVisualization_RepaintNode.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Objects_AbstractWrapper_InitPorts.htm`

teten Klassen implementiert werden muss. Diese Methode stellt die polymorphe Initialsierungsphase dar (vgl. *Unterabschnitt 3.3.3 Polymorphe Brückenknoten* und *Unterabschnitt 4.3.4 Polymorphie der Brückenknoten*).

AbstractMethodBase

Die abstrakte Klasse `DynamicNode.Ext.Objects.AbstractMethodBase`[1] (vgl. *Abbildung A.8* im Anhang auf Seite 154 und *Listing A.11* im Anhang auf Seite 165) ist die Basisklasse für alle voll polymorphen Brückenknoten (vgl. *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*). Dazu gehören die Brückenknoten für Methodenaufrufe und der Brückenknoten für den Aufruf von Konstruktoren. Diese Klasse befindet sich in dem Assembly `DNExtObjects` (vgl. *Abbildung 5.1*).

Diese Klasse bietet die Eigenschaft `MethodBase MethodBase { get; set; }`. Diese Eigenschaft ermöglicht das Setzen der Adresse, der zu bindenden Interaktion, in Form eines `MethodBase`-Objektes. Im Setter der Eigenschaft wird die Methode `void InitPorts()` aufgerufen und somit die polymorphe Initialisierungsphase eingeleitet. Da in den Kindklassen die Adresse nicht als `MethodBase`-Objekt, sondern in einer spezialisierten Form benötigt wird, definiert `AbstractMethodBase` die abstrakte Methode `void CheckMethodBase(MethodBase)`[2], welche im Setter der Eigenschaft `MethodBase` aufgerufen wird. Kindklassen müssen `CheckMethodBase()` implementieren und können eine Ausnahme werfen, wenn die übergebene Adresse nicht vom erwarteten Typ ist.

Zur Anpassung der Visualisierung überschreibt `AbstractMethodBase` die beiden Methoden `GetInteractionSymbol1()` und `GetInteractionSymbol2()`. Die erste gibt das Symbol für eine objektgebundene Methode zurück, die zweite gibt `null` zurück. Auch die Methode `GetTextLine1()` wird implementiert. Diese Methode ermittelt aus der Adresse der gebundenen Interaktion mit Hilfe von `MethodBase.ReflectedType.Namespace` den Namensraum der Klasse, für die die Interaktion definiert ist, und gibt ihn zurück. Falls keine Interaktion gebunden ist, gibt sie `null` zurück.

Der wichtigste Teil dieser Klasse besteht aus den zwei Methoden `void InitPorts()` und `void Work(TokenSet)`[3].

InitPorts – die polymorphe Initialsierungsphase Die Implementierung der Methode `InitPorts()` folgt bis auf kleine Abweichungen dem Algorithmus in *Konfigurationsalgorithmus für voll polymorphe Brückenknoten* auf Seite 71. Die Besonderheit ist, dass der Algorith-

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_AbstractMethodBase.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Objects_AbstractMethodBase_CheckMethodBase.htm`
[3] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Core_StreamNode_Work_1.htm`

mus hier für statische, objektgebundene Methoden und Konstruktoren verallgemeinert ist. Um die Verallgemeinerung zu ermöglichen, definiert die Klasse `AbstractMethodBase` drei abstrakte Eigenschaften:

- `Type ReturnType { get; }`
- `string ReturnOutPortId { get; }`
- `bool IsMethodStatic { get; }`

Unter Abfrage dieser Eigenschaften verzweigt die Methode `InitPorts`, um für Interaktionsformen bei denen `ReturnType` nicht gleich `null` ist, einen passenden Ausgang anzulegen oder um für Interaktionsformen, bei denen `IsMethodStatic` gleich `false` ist, einen Eingang für den Objektkontext zu erzeugen.

Work – die Knotenoperation Die Methode `Work()` wird von der Laufzeitumgebung von DynamicNodes aufgerufen, wenn der Knoten aktiviert wird. Bei den polymorphen Brückenknoten besteht die Knotenoperation in der Ausführung der Interaktion (vgl. *Ausführungsphase* auf Seite 73). `AbstractMethodBase` überschreibt diese Methode. Die Implementierung folgt dem Schema aus *Ausführungsphase* auf Seite 73, wobei der Algorithmus, ebenso wie die Initialisierungsphase für objektgebundene Methoden, statische Methoden und Konstruktoren, verallgemeinert ist.

Es werden alle Parameter mit `ParameterCollection MethodBase.GetParameters()` ermittelt und ein `Object`-Array für die Parameterwerte mit der entsprechenden Länge vorbereitet. Anschließend werden die Parameter durchlaufen und die Werte von den entsprechenden Eingängen gelesen. Ist die Interaktion nicht statisch (existiert ein Eingang *instance* für den Objektkontext), so wird auch der Objektkontext für die Interaktion von *instance* gelesen, andernfalls wird die Variable für die Speicherung des Objektkontexts auf `null` belassen. Ist der Objektkontext `null`, die Interaktion ist jedoch ein objektgebundener Methodenaufruf, wird die Ausführung der Interaktion abgebrochen.

Nun wird die Ausführung verzweigt, in Abhängigkeit davon, ob die Adresse der Interaktion (`MethodBase`) ein `ConstructorInfo`-Objekt ist, und demzufolge ein Konstruktor gebunden ist, oder nicht. Falls ein Konstruktor gebunden ist, wird `Invoke()` ohne Objektkontext aufgerufen. Ist die Adresse ein MethodInfo-Objekt, wird die Methode `Invoke()` mit dem aktuellen Objektkontext aufgerufen. Falls die Methode statisch ist, wird `null` als Objektkontext übergeben.

Falls die Interaktion einen Rückgabewert besitzt, bei Konstruktoren ist dies immer der Fall, wird der Rückgabewert des Aufrufs von `Invoke()` über den Ausgang *return* weitergegeben.

5.2.2 Brückenknoten für Methoden

In diesem Unterabschnitt werden die Klassen der Brückenknoten für den objektgebundenen Methodenaufruf (`NMethod`) und den statischen Methodenaufruf (`NStaticMethod`) erläutert. Beide Klassen erben von der Basisklasse `AbstractMethod` (vgl. *Abbildung A.9* im Anhang auf Seite 155). Alle drei Klassen befinden sich in dem Assembly `NLibObjects` (vgl. *Abbildung 5.1*).

AbstractMethod

Die Klasse `DynamicNode.Lib.Objects.AbstractMethod` (vgl. *Listing A.12* im Anhang auf Seite 169) erbt von `AbstractMethodBase` und spezialisiert sie für den Aufruf von Methoden. Dazu implementiert sie die Eigenschaft `ReturnOutPortId` und gibt über sie immer „`return`" als Name des Ausgangs für Rückgabewerte zurück. Ebenso implementiert sie die Eigenschaft `ReturnType` und gibt den Rückgabedatentyp der aktuell gebundenen Methode (`MethodInfo.ReturnType`) oder `null` zurück, falls keine Methode gebunden ist.

`AbstractMethod` deklariert eine Eigenschaft `MethodInfo Method { get; set; }`, welche den Wert aus der geerbten Eigenschaft `MethodBase` durchreicht und die Typenumwandlung von und in die Klasse `MethodInfo` durchführt. Durch diese Eigenschaft steht den Kindklassen `NMethod` und `NStaticMethod` die aktuelle Adresse der gebundenen Methode direkt in Form eines `MethodInfo`-Objektes zur Verfügung.

Zur Anpassung der Visualisierung implementiert `AbstractMethod` die Methode `GetTextLine2()`. Diese Methode ermittelt den Namen der Klasse, für die die aktuell gebundene Methode definiert ist, und den Namen der Methode und gibt beides mit einem Punkt verbunden zurück.

Eine wichtige Aufgabe der Klasse `AbstractMethod` ist das Überschreiben der Methoden `OnStore()` und `OnRestore()`. In `OnStore()` wird die Adresse in eine Zeichenkette umgewandelt und als Konfigurationsparameter mit dem Namen „`method`" (z. B. für die Speicherung in einer XML-Datei) gesichert. Die Methode `OnRestore()` sucht in der übergebenen Gruppe von Konfigurationsparametern nach der Einstellung mit dem Namen „`method`" und versucht mit dem hinterlegten Wert das `MethodInfo`-Objekt als Adresse zu rekonstruieren, um anschließend die Bindung an die adressierte Methode auszulösen.

Für beide Arten von Methoden, statisch und objektgebunden, implementiert `AbstractMethod` die Knotensteuerung zur Auswahl der Adresse von Klasse und Methode (vgl. *Benutzeroberfläche* auf Seite 55 und *Initialisierungsphase* auf Seite 69). Dazu überschreibt sie die Methode `VisualNode.InitControlPanel(NodeControlPanelExtension)`[1] und definiert eine virtuelle Methode `void InitSelector(MethodSelector)`, über den die Klassen `NMethod`

[1] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Visual_VisualNode_InitControlPanel.htm`

und `NStaticMethod` die Gelegenheit erhalten, die Knotensteuerung – `MethodSelector`[1] – zur Auswahl der Adresse anzupassen.

NMethod

Die Klasse `DynamicNode.Lib.Objects.NMethod` (vgl. *Listing A.13* im Anhang auf Seite 171) erbt von `AbstractMethod` und spezialisiert sie für den Aufruf von objektgebundenen Methoden. Dazu implementiert sie die Methode `AbstractMethodBase.CheckMethodBase()`[2] und wirft eine Ausnahme, falls die übergebene Adresse nicht auf eine objektgebundene Methode verweist. Auch implementiert `NMethod` die Eigenschaft `IsMethodStatic` und gibt `false` zurück.

Die Methode `AbstractMethod.InitSelector()` wird überschrieben und darin die übergebene Knotensteuerung so konfiguriert, dass sie nur objektgebundene Methoden zur Auswahl stellt.

In *Abbildung 5.2* ist die Visualisierung von einer `NMethod`-Instanz dargestellt.

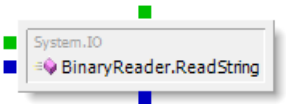


Abbildung 5.2: Visualisierung des Brückenknotens `NMethod`

NStaticMethod

Die Klasse `DynamicNode.Lib.Objects.NStaticMethod` (vgl. *Listing A.14* im Anhang auf Seite 172) erbt von `AbstractMethod` und spezialisiert sie für den Aufruf von statischen Methoden. Dazu implementiert sie die Methode `CheckMethodBase()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf eine statische Methode verweist. Auch implementiert `NStaticMethod` die Eigenschaft `IsMethodStatic` und gibt `true` zurück.

Auch die Methode `AbstractMethod.InitSelector()` wird überschrieben und darin die übergebene Knotensteuerung so konfiguriert, dass sie nur statische Methoden zur Auswahl stellt. Zur Anpassung der Visualisierung wird die Methode `GetInteractionSymbol1()` überschrieben und das Symbol für einen statischen Methodenaufruf zurückgegeben.

In *Abbildung 5.3* ist die Visualisierung von einer `NStaticMethod`-Instanz dargestellt.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_MethodSelector.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Ext_Objects_AbstractMethodBase_CheckMethodBase.htm`

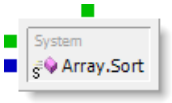


Abbildung 5.3: Visualisierung des Brückenknotens `NStaticMethod`

5.2.3 Brückenknoten für Konstruktoren

Die Klasse `DynamicNode.Lib.Objects.NConstructor` (vgl. *Abbildung A.9* im Anhang auf Seite 155 und *Listing A.15* im Anhang auf Seite 173) erbt von `AbstractMethodBase` und spezialisiert sie für den Aufruf von Konstruktoren (vgl. *Unterabschnitt 2.1.4 Objektorientierte Programmierung*). Die Klasse `NConstructor` befindet sich in dem Assembly `NLibObjects` (vgl. *Abbildung 5.1*). Dazu implementiert sie die Methode `CheckMethodBase()` und wirft eine Ausnahme, wenn die übergebene Adresse nicht auf einen Konstruktor verweist. Sie implementiert auch die Eigenschaften `ReturnType` und `ReturnOutPortId`. `ReturnType` gibt die Klasse zurück, für die der aktuell gebundene Konstruktor definiert ist oder `null`, falls kein Konstruktor gebunden ist. `ReturnOutPortId` gibt als Name für den Ausgang die Zeichenkette „`instance`" zurück. Auch die Eigenschaft `IsMethodStatic` wird implementiert und gibt immer `true` zurück.

`NConstructor` definiert eine Eigenschaft `ConstructorInfo Constructor { get; set; }`, welche die Adresse von `MethodBase` durchreicht und die Typenumwandlung von und in `ConstructorInfo` vornimmt. Mit dieser Eigenschaft steht die Adresse für den Konstruktor in der Klasse `NConstructor` direkt in Form eines `ConstructorInfo`-Objektes zur Verfügung.

Für die Anpassung der Visualisierung des Brückenknotens wird Methode `GetTextLine2()` implementiert und die Methode `GetInteractionSymbol1()` berschrieben. `GetTextLine2()` gibt eine Zeichenkette zurück, die sich aus „`new␣`" und dem Namen der Klasse zusammensetzt, für die der gebundene Konstruktor definiert ist. `GetInteractionSymbol1()` gibt das Symbol für einen Konstruktoraufruf zurück.

Die Methoden `OnStore()` und `OnRestore()` werden nach dem gleichen Muster wie `AbstractMethod` überschrieben (vgl. *Unterabschnitt 5.2.2 Brückenknoten für Methoden*). Dabei wird die Adresse des gebundenen Konstruktors als einziger Konfigurationsparameter gesichert und wiederhergestellt.

Die Implementierung für die Knotensteuerung, zur Auswahl der Adresse für den Konstruktor, folgt nahezu vollständig dem Muster aus `AbstractMethod` – mit der Abweichung, dass es keine virtuelle Methode `InitSelector()` gibt. Denn die Knotensteuerung `ConstructorSelector`[1] wird nur für `NConstructor` verwendet und braucht dementsprechend nicht durch Kindklassen angepasst zu werden.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_ConstructorSelector.htm`

5.2.4 Brückenknoten für Felder

In diesem Unterabschnitt werden die Klassen der Brückenknoten für das Lesen und Schreiben von objektgebundenen und statischen Feldern erläutert (`NRead`, `NWrite`, `NStaticRead` und `NStaticWrite`) (vgl. *Unterabschnitt 2.1.2 Imperative Programmierung*). Die Klassen erben von der Basisklasse `AbstractField` (vgl. *Abbildung A.11* im Anhang auf Seite 157). Alle fünf Klassen befinden sich in dem Assembly `NLibObjects` (vgl. *Abbildung 5.1*).

AbstractField

Die Klasse `DynamicNode.Lib.Objects.AbstractField` (vgl. *Listing A.16* im Anhang auf Seite 176) erbt von `AbstractWrapper` und spezialisiert sie für das Lesen und Schreiben von Feldern.

Diese Klasse bietet die Eigenschaft `FieldInfo Field { get; set; }`. Diese Eigenschaft ermöglicht das Setzen der Adresse des zu bindenden Feldes in Form eines `FieldInfo`-Objektes. Im Setter der Eigenschaft wird die Methode `void InitPorts()` aufgerufen und somit die polymorphe Initialisierungsphase eingeleitet. Damit ein Brückenknoten für statische Felder nicht an ein objektgebundenes Feld oder ein Brückenknoten für objektgebundene Felder nicht an ein statisches Feld gebunden werden kann, definiert `AbstractField` die abstrakte Methode `void CheckFieldInfo(FieldInfo)`, welche im Setter der Eigenschaft `Field` aufgerufen wird, als Kontrollmechanismus. Kindklassen müssen `CheckFieldInfo()` implementieren und können eine Ausnahme werfen, wenn die übergebene Adresse nicht auf ein Feld mit den erforderlichen Eigenschaften verweist.

Zur Anpassung der Visualisierung implementiert `AbstractField` die Methoden `GetTextLine1()` und `GetTextLine2()` und überschreibt die Methode `GetInteractionSymbol1()`. `GetTextLine1()` ermittelt mit `FieldInfo.ReflectedType.Namespace` den Namensraum der Klasse, für die das aktuell gebundene Feld definiert ist und gibt ihn zurück. `GetTextLine2()` ermittelt den Namen der Klasse, für die das aktuell gebundene Feld definiert ist, und den Namen des gebundenen Feldes und gibt beides mit einem Punkt als Verbindungszeichen zurück. Falls kein Feld gebunden ist, geben beide `GetTextLine`-Methoden `null` zurück. Die Methode `GetInteractionSymbol1()` gibt das Symbol für ein objektgebundenes Feld zurück.

Die Methoden `OnStore()` und `OnRestore()` werden nach dem gleichen Muster wie in der Basisklasse `AbstractMethod` überschrieben (vgl. *AbstractMethod* auf Seite 91). Auch die Bereitstellung der Knotensteuerung, für die Auswahl der Adresse des zu bindenden Feldes, folgt genau dem Muster in `AbstractMethod`, wobei als Knotensteuerung das Steuerelement `FieldSelector`[1] zum Einsatz kommt.

[1] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_FieldSelector.htm`

NRead

Die Klasse `DynamicNode.Lib.Objects.NRead` (vgl. *Listing A.17* im Anhang auf Seite 178) erbt von `AbstractField` und spezialisiert sie für das Lesen von objektgebundenen Feldern. Sie implementiert die Methode `AbstractField.CheckFieldInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf ein objektgebundenes Feld verweist.

In der Initialisierungsphase des Brückenknotens, repräsentiert durch die Methode `void Init()`, die während der Instanzierung ausgeführt wird, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Lesen eines objektgebundenen Feldes notwendig sind. Dazu gehören *instance* als Eingang für den Objektkontext, *value* als Ausgang für den gelesenen Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NRead` wird die Method `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()`, der polymorphen Initialsierungsphase, wird lediglich die Kompatibilität der Anschlüsse an die Datentypen von Objektkontext und Feld angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Dazu liest sie in Schritt 1 den Objektkontext vom Eingang *instance*. Falls der Objektkontext `null` ist, wird die Ausführung an dieser Stelle abgebrochen. In Schritt 2 ruft sie `Object FieldInfo.GetValue(Object)`[1] auf und übergibt dabei den Objektkontext. Der Rückgabewert von `GetValue()` wird zwischengespeichert. In Schritt 3 wird über den Ausgang *value* der zwischengespeicherte Wert weitergegeben. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NRead` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Lesen eines Wertes zurück. Ist das gebundene Feld schreibgeschützt, wird `null` zurückgegeben, weil in diesem Fall nicht zwischen Lesen und Schreiben unterschieden werden muss.

Zur Konfiguration der Knotensteuerung implementiert `NRead` die Methode `AbstractField.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur objektgebundene Felder zur Auswahl stellt.

In *Abbildung 5.4* ist die Visualisierung von einer `NRead`-Instanz dargestellt.

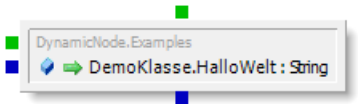


Abbildung 5.4: Visualisierung des Brückenknotens `NRead`

[1] `http://msdn.microsoft.com/de-de/library/system.reflection.fieldinfo.getvalue.aspx`

NWrite

Die Klasse `DynamicNode.Lib.Objects.NWrite` (vgl. *Listing A.18* im Anhang auf Seite 180) erbt von `AbstractField` und spezialisiert sie für das Schreiben von objektgebundenen Feldern. Sie implementiert die Methode `AbstractField.CheckFieldInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf ein objektgebundenes Feld verweist, das geschrieben werden kann.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Schreiben eines objektgebundenen Feldes notwendig sind. Dazu gehören *instance* als Eingang für den Objektkontext, *value* als Eingang für den zu schreibenden Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NWrite` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität der Anschlüsse an die Datentypen von Objektkontext und Feld angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Dazu liest sie in Schritt 1 den Objektkontext vom Eingang *instance* und den neuen Wert für das Feld vom Eingang *value*. Falls der Objektkontext `null` ist, wird die Ausführung an dieser Stelle abgebrochen. In Schritt 2 ruft sie `void FieldInfo.SetValue(Object, Object)`[1] auf und übergibt dabei den Objektkontext und den neuen Wert. Der Schritt 3 ist für `NWrite` leer, da es keine Rückgabewerte gibt. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NWrite` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Schreiben eines Wertes zurück.

Zur Konfiguration der Knotensteuerung implementiert `NWrite` die Methode `AbstractField.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur objektgebundene Felder zur Auswahl stellt, die geschrieben werden dürfen.

In 5.5 ist die Visualisierung von einer `NWrite`-Instanz dargestellt.

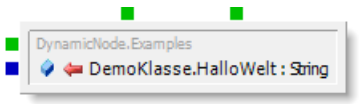


Abbildung 5.5: Visualisierung des Brückenknotens `NWrite`

NStaticRead

Die Klasse `DynamicNode.Lib.Objects.NStaticRead` (vgl. *Listing A.19* im Anhang auf Seite 182) erbt von `AbstractField` und spezialisiert sie für das Lesen von statischen Feldern. Sie

[1] `http://msdn.microsoft.com/de-de/library/6z33zd7h.aspx`

implementiert die Methode `AbstractField.CheckFieldInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf ein statisches Feld verweist.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Lesen eines statischen Feldes notwendig sind. Diese sind *value* als Ausgang für den gelesenen Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NStaticRead` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität des Ausgangs *value* an den Datentyp des gebundenen Feldes angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Der Schritt 1 ist für `NStaticRead` leer, da keine Eingabedaten verarbeitet werden. In Schritt 2 ruft sie `Object FieldInfo.GetValue(Object)`[1] auf und übergibt dabei `null` als Objektkontext. Der Rückgabewert von `GetValue()` wird zwischengespeichert. In Schritt 3 wird über den Ausgang *value* der zwischengespeicherte Wert weitergegeben. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NStaticRead` die Methode `GetInteractionSymbol1()`. Welches Symbol in dieser Klasse zurückgegeben wird, hängt von dem gebundenen Feld ab. Ist das Feld eine Konstante (statisch und schreibgeschützt), wird das Symbol für eine Konstante zurückgegeben. Ist das Symbol eine Konstante und ist die Klasse, für welche die Konstante definiert ist, eine Aufzählung (*Enumeration*), so wird das Symbol für ein Aufzählungselement zurückgegeben. Nur wenn diese Bedingungen nicht erfüllt sind, wird das Symbol für ein statisches Feld zurückgegeben. Auch die Methode `GetInteractionSymbol2()` wird überschrieben. Diese gibt genau dann ein Symbol für das Lesen eines Wertes zurück, wenn `GetInteractionSymbol1()` das Symbol für ein statisches Feld zurückgibt. Andernfalls wird `null` zurückgegeben.

Zur Konfiguration der Knotensteuerung implementiert `NStaticRead` die Methode `AbstractField.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur statische Felder zur Auswahl stellt.

In 5.6 ist die Visualisierung von einer `NStaticRead`-Instanz dargestellt.

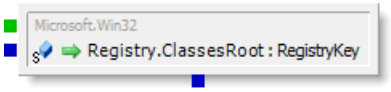


Abbildung 5.6: Visualisierung des Brückenknotens `NStaticRead`

[1] `http://msdn.microsoft.com/de-de/library/system.reflection.fieldinfo.getvalue.aspx`

NStaticWrite

Die Klasse `DynamicNode.Lib.Objects.NStaticWrite` (vgl. *Listing A.20* im Anhang auf Seite 184) erbt von `AbstractField` und spezialisiert sie für das Schreiben von statischen Feldern. Sie implementiert die Methode `AbstractField.CheckFieldInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf ein statisches Feld verweist, das geschrieben werden kann.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Schreiben eines statischen Feldes notwendig sind. Das sind *value* als Eingang für den zu schreibenden Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NStaticWrite` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität des Eingangs *value* an den Datentyp des gebundenen Feldes angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Dazu liest sie in Schritt 1 den neuen Wert für das Feld vom Eingang *value*. In Schritt 2 ruft sie `void FieldInfo.SetValue(Object, Object)`[1] auf und übergibt dabei `null` als Objektkontext und den neuen Wert für das Feld. Der Schritt 3 ist für `NStaticWrite` leer, da es keine Rückgabewerte gibt. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NStaticWrite` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Schreiben eines Wertes zurück. Zusätzlich wird auch die Methode `GetInteractionSymbol1()` überschrieben, welche in dieser Klasse immer das Symbol für ein statisches Feld zurückgibt.

Zur Konfiguration der Knotensteuerung implementiert `NStaticWrite` die Methode `AbstractField.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur statische Felder zur Auswahl stellt, die geschrieben werden dürfen.

In *Abbildung 5.7* ist die Visualisierung von einer `NStaticWrite`-Instanz dargestellt.

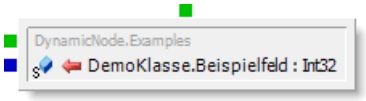


Abbildung 5.7: Visualisierung des Brückenknotens `NStaticWrite`

5.2.5 Brückenknoten für Eigenschaften

In diesem Unterabschnitt werden die Klassen der Brückenknoten für das Lesen und Schreiben von objektgebundenen und statischen Eigenschaften erläutert (`NGet`, `NSet`, `NStaticGet` und

[1] `http://msdn.microsoft.com/de-de/library/6z33zd7h.aspx`

NStaticSet) (vgl. *Eigenschaften* auf Seite 23). Die Klassen erben von der Basisklasse AbstractProperty (vgl. *Abbildung A.13* im Anhang auf Seite 159). Alle fünf Klassen befinden sich in dem Assembly NLibObjects (vgl. *Abbildung 5.1*).

Die Implementierung der Brückenknoten für Eigenschaften ähnelt sehr stark der Implementierung der Brückenknoten für Felder (vgl. *Unterabschnitt 5.2.4 Brückenknoten für Felder*).

AbstractProperty

Die Klasse DynamicNode.Lib.Objects.AbstractProperty (vgl. *Listing A.21* im Anhang auf Seite 186) erbt von AbstractWrapper und spezialisiert sie für das Lesen und Schreiben von Eigenschaften.

Diese Klasse bietet die Eigenschaft PropertyInfo Property { get; set; }. Diese Eigenschaft ermöglicht das Setzen der Adresse, der zu bindenden Eigenschaft, in Form eines PropertyInfo-Objektes. Im Setter der Eigenschaft wird die Methode void InitPorts() aufgerufen und somit die polymorphe Initialisierungsphase eingeleitet. Analog zu AbstractField definiert AbstractProperty die abstrakte Methode void CheckPropertyInfo(PropertyInfo) als Kontrollmechanismus, welche im Setter der Eigenschaft Property aufgerufen wird. Kindklassen müssen CheckPropertyInfo() implementieren und können eine Ausnahme werfen, wenn die übergebene Adresse nicht auf eine .NET-Eigenschaft mit den erforderlichen Eigenschaften verweist.

Zur Anpassung der Visualisierung implementiert AbstractProperty die Methoden GetTextLine1(), GetTextLine2() und überschreibt die Methode GetInteractionSymbol1(). GetTextLine1() ermittelt mit PropertyInfo.ReflectedType.Namespace den Namensraum der Klasse, für die die aktuell gebundene Eigenschaft definiert ist, und gibt ihn zurück. GetTextLine2() ermittelt den Namen der Klasse, für die die aktuell gebundene Eigenschaft definiert ist, und den Namen der gebundenen Eigenschaft und gibt beides mit einem Punkt als Verbindungszeichen zurück. Falls keine Eigenschaft gebunden ist, geben beide GetTextLine-Methoden null zurück. Die Methode GetInteractionSymbol1() gibt das Symbol für eine objektgebundene Eigenschaft zurück.

Die Methoden OnStore() und OnRestore() werden nach dem gleichen Muster wie in der Basisklasse AbstractMethod überschrieben (vgl. *AbstractMethod* auf Seite 91). Auch die Bereitstellung der Knotensteuerung, für die Auswahl der Adresse des zu bindenden Feldes, folgt genau dem Muster in AbstractMethod, wobei als Knotensteuerung das Steuerelement PropertySelector[1] zum Einsatz kommt.

[1] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_PropertySelector.htm

NGet

Die Klasse `DynamicNode.Lib.Objects.NGet` (vgl. *Listing A.22* im Anhang auf Seite 189) erbt von `AbstractProperty` und spezialisiert sie für das Lesen von objektgebundenen Eigenschaften. Sie implementiert die Methode `AbstractProperty.CheckPropertyInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf eine objektgebundene Eigenschaft verweist, die gelesen werden kann.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Lesen einer objektgebundenen Eigenschaft notwendig sind. Dazu gehören *instance* als Eingang für den Objektkontext, *value* als Ausgang für den gelesenen Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NGet` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität der Anschlüsse an die Datentypen von Objektkontext und Eigenschaft angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Dazu liest sie in Schritt 1 den Objektkontext vom Eingang *instance*. Falls der Objektkontext `null` ist, wird die Ausführung an dieser Stelle abgebrochen. In Schritt 2 ruft sie `Object PropertyInfo.GetValue(Object, Object[])`[1] auf und übergibt dabei den Objektkontext. Der zweite Parameter nimmt ein Array mit Index-Parametern entgegen. Index-Parameter werden in dieser Arbeit nicht berücksichtigt. Für diesen Parameter wird ein leeres `Object`-Array übergeben. Der Rückgabewert von `GetValue()` wird zwischengespeichert. In Schritt 3 wird über den Ausgang *value* der zwischengespeicherte Wert weitergegeben. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NGet` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Lesen eines Wertes zurück. Auch die Methode `GetInteractionSymbol1()` wird überschrieben. Sie gibt normalerweise das Symbol für eine objektgebundene Eigenschaft zurück. Ist die gebundene Eigenschaft jedoch schreibgeschützt, wird das Symbol für eine schreibgeschützte objektgebundene Eigenschaft zurückgegeben.

Zur Konfiguration der Knotensteuerung implementiert `NGet` die Methode `AbstractProperty.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur objektgebundene Eigenschaften zur Auswahl stellt, die gelesen werden können.

In *Abbildung 5.8* ist die Visualisierung von einer `NGet`-Instanz dargestellt.

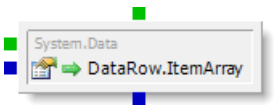


Abbildung 5.8: Visualisierung des Brückenknotens `NGet`

[1] `http://msdn.microsoft.com/de-de/library/b05d59ty.aspx`

NSet

Die Klasse `DynamicNode.Lib.Objects.NSet` (vgl. *Listing A.23* im Anhang auf Seite 191) erbt von `AbstractProperty` und spezialisiert sie für das Schreiben von objektgebundenen Eigenschaften. Sie implementiert die Methode `AbstractProperty.CheckPropertyInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf eine objektgebundene Eigenschaft verweist, die geschrieben werden kann.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Schreiben einer objektgebundenen Eigenschaft notwendig sind. Dazu gehören *instance* als Eingang für den Objektkontext, *value* als Eingang für den zu schreibenden Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NSet` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität der Anschlüsse an die Datentypen von Objektkontext und Eigenschaft angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Dazu liest sie in Schritt 1 den Objektkontext vom Eingang *instance* und den neuen Wert für die Eigenschaft vom Eingang *value*. Falls der Objektkontext `null` ist, wird die Ausführung an dieser Stelle abgebrochen. In Schritt 2 ruft sie `void PropertyInfo.SetValue(Object, Object, Object[])`[1] auf und übergibt dabei den Objektkontext und den neuen Wert. Der dritte Parameter nimmt ein Array mit Index-Parametern entgegen. Index-Parameter werden in dieser Arbeit nicht berücksichtigt. Für diesen Parameter wird ein leeres `Object`-Array übergeben. Der Schritt 3 ist für `NSet` leer, da es keine Rückgabewerte gibt. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NSet` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Schreiben eines Wertes zurück.

Zur Konfiguration der Knotensteuerung implementiert `NSet` die Methode `AbstractProperty.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur objektgebundene Eigenschaften zur Auswahl stellt, die geschrieben werden dürfen.

In 5.9 ist die Visualisierung von einer `NSet`-Instanz dargestellt.

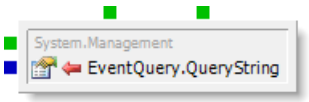


Abbildung 5.9: Visualisierung des Brückenknotens `NSet`

[1] `http://msdn.microsoft.com/de-de/library/xb5dd1f1.aspx`

NStaticGet

Die Klasse `DynamicNode.Lib.Objects.NStaticGet` (vgl. *Listing A.24* im Anhang auf Seite 193) erbt von `AbstractProperty` und spezialisiert sie für das Lesen von statischen Eigenschaften. Sie implementiert die Methode `AbstractProperty.CheckPropertyInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf eine statische Eigenschaft verweist, die gelesen werden kann.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Lesen einer statischen Eigenschaft notwendig sind. Diese sind *value* als Ausgang für den gelesenen Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NStaticGet` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität des Ausgangs *value* an den Datentyp der gebundenen Eigenschaft angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Der Schritt 1 ist für `NStaticGet` leer, da keine Eingabedaten verarbeitet werden. In Schritt 2 ruft sie `Object PropertyInfo.GetValue(Object, Object[])`[1] auf und übergibt dabei `null` als Objektkontext. Der zweite Parameter nimmt ein Array mit Index-Parametern entgegen. Index-Parameter werden in dieser Arbeit nicht berücksichtigt. Für diesen Parameter wird auch ein leeres `Object`-Array übergeben. Der Rückgabewert von `GetValue()` wird zwischengespeichert. In Schritt 3 wird über den Ausgang *value* der zwischengespeicherte Wert weitergegeben. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NStaticGet` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Lesen eines Wertes zurück. Auch die Methode `GetInteractionSymbol1()` wird überschrieben. Sie gibt normalerweise das Symbol für eine statische Eigenschaft zurück, ist die gebundene Eigenschaft jedoch schreibgeschützt, wird das Symbol für eine schreibgeschützte statische Eigenschaft zurückgegeben.

Zur Konfiguration der Knotensteuerung implementiert `NStaticRead` die Methode `AbstractField.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur statische Eigenschaften zur Auswahl stellt, die gelesen werden können.

In *Abbildung 5.10* ist die Visualisierung von einer `NStaticGet`-Instanz dargestellt.

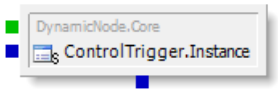


Abbildung 5.10: Visualisierung des Brückenknotens `NStaticGet`

[1] `http://msdn.microsoft.com/de-de/library/b05d59ty.aspx`

NStaticSet

Die Klasse `DynamicNode.Lib.Objects.NStaticSet` (vgl. *Listing A.25* im Anhang auf Seite 195) erbt von `AbstractProperty` und spezialisiert sie für das Schreiben von statischen Eigenschaften. Sie implementiert die Methode `AbstractProperty.CheckPropertyInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf eine statische Eigenschaft verweist, die geschrieben werden kann.

In der Initialisierungsphase des Brückenknotens, `void Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Schreiben einer statischen Eigenschaft notwendig sind. Das sind *value* als Eingang für den zu schreibenden Wert sowie *triggerIn* und *triggerOut* als Anschlüsse für den Kontrollfluss.

Von `NStaticSet` wird die Methode `InitPorts()` implementiert und die Methode `Work()` überschrieben. In `InitPorts()` wird lediglich die Kompatibilität des Eingangs *value* an den Datentyp der gebundenen Eigenschaft angepasst. Die Methode `Work()` führt die Schritte nach *Ausführungsphase* auf Seite 73 aus. Dazu liest sie in Schritt 1 den neuen Wert für die Eigenschaft vom Eingang *value*. In Schritt 2 ruft sie `void PropertyInfo.SetValue(Object, Object, Object[])`[1] auf und übergibt dabei `null` als Objektkontext und den neuen Wert für das Feld. Der dritte Parameter nimmt ein Array mit Index-Parametern entgegen. Index-Parameter werden in dieser Arbeit nicht berücksichtigt. Für diesen Parameter wird ein leeres `Object`-Array übergeben. Der Schritt 3 ist für `NStaticSet` leer, da es keine Rückgabewerte gibt. In Schritt 4 wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben.

Zur Anpassung der Visualisierung überschreibt `NStaticSet` die Methode `GetInteractionSymbol2()` und gibt das Symbol für das Schreiben eines Wertes zurück. Zusätzlich wird auch die Methode `GetInteractionSymbol1()` überschrieben, welche in dieser Klasse immer das Symbol für eine statische Eigenschaft zurückgibt.

Zur Konfiguration der Knotensteuerung implementiert `NStaticSet` die Methode `AbstractProperty.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur statische Eigenschaften zur Auswahl stellt, die geschrieben werden dürfen.

In 5.11 ist die Visualisierung von einer `NStaticSet`-Instanz dargestellt.

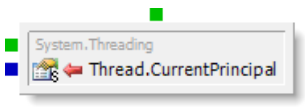


Abbildung 5.11: Visualisierung des Brückenknotens `NStaticSet`

[1] `http://msdn.microsoft.com/de-de/library/xb5dd1f1.aspx`

5.2.6 Brückenknoten für Ereignisse

In diesem Unterabschnitt werden die Klassen der Brückenknoten für das Überwachen von objektgebundenen und statischen Ereignissen erläutert (`NEvent` und `NStatic`Event) (vgl. *Ereignisse* auf Seite 25). Die Klassen erben von der Basisklasse `AbstractEvent` (vgl. *Abbildung A.15* im Anhang auf Seite 160 und *Abbildung A.15* im Anhang auf Seite 160). Alle drei Klassen befinden sich in dem Assembly `NLibObjects` (vgl. *Abbildung 5.1*).

Für das Überwachen von Ereignissen wird ein Handler benötigt, der bei dem zu überwachenden Ereignis registriert werden kann. Dieser wird jedesmal aufgerufen, wenn das Ereignis auftritt. Die Registrierung und Deregistrierung erfolgt durch den Aufruf zweier Methoden (vgl. *Ereignisse* auf Seite 25). Ein Datenflussgraph bietet keine Möglichkeit einen solchen Handler zu definieren. Deshalb ist es auch nicht sinnvoll, die Methoden zum Registrieren und Deregistrieren als Methodenaufrufe in einen Datenflussgraphen einzubinden. Was eigentlich von einem Ereignis in einem Datenflusssystem erwartet wird, ist ein Ausgang, der immer dann eine Marke an verbundene Eingänge weitergibt, wenn das Ereignis ausgelöst wurde.

Trifft ein Objekt über den Eingang im Brückenknoten ein, registriert dieser eine interne Handler-Methode bei dem Ereignis im Kontext des eingetroffenen Objektes. Wird der Handler durch das Ereignis aufgerufen, erzeugt er eine neue Ereignis-Marke und leitet diese über den Ausgang des Knotens weiter. Trifft erneut ein Objekt über den Eingang ein, wird der Handler zunächst bei dem Ereignis im Kontext des alten Objektes abgemeldet und anschließend wieder bei dem Ereignis im Kontext des neuen Objektes angemeldet.

AbstractEvent

Die Klasse `DynamicNode.Lib.Objects.AbstractEvent` (vgl. *Listing A.26* im Anhang auf Seite 197) erbt von `AbstractWrapper` und spezialisiert sie für das Überwachen von Ereignissen.

Diese Klasse bietet die Eigenschaft `EventInfo Event { get; set; }`. Diese Eigenschaft ermöglicht das Setzen der Adresse des zu bindenden Ereignisses in Form eines `EventInfo`-Objektes. Im Setter der Eigenschaft wird die Methode `void InitPorts()` aufgerufen und somit die polymorphe Initialisierungsphase eingeleitet. Analog zu `AbstractField`, definiert `AbstractProperty` die abstrakte Methode `void CheckEventInfo(EventInfo)` als Kontrollmechanismus, welche im Setter der Eigenschaft `Event` aufgerufen wird. Kindklassen müssen `CheckEventInfo()` implementieren und können eine Ausnahme werfen, wenn die übergebene Adresse nicht auf ein Ereignis mit den erforderlichen Eigenschaften verweist.

Die Klasse `AbstractEvent` implementiert wenigstens die Funktionalität, die für die Überwachung von statischen Ereignissen erforderlich ist und an einigen Stellen auch Funktionalität

für die Überwachung von objektgebundenen Ereignissen. Dadurch kann die Implementierung der Kindklasse `NStaticEvent` sehr kurz ausfallen, während die Implementierung von `NEvent` Funktionalität für den Objektkontext hinzufügen muss.

In der Initialisierungsphase des Brückenknotens, `Init()`, werden alle Ein- und Ausgänge erzeugt, die für den Brückenknoten zum Überwachen von statischen Ereignissen notwendig sind. Das sind *sender* als Ausgang für das Objekt, welches das Ereignis ausgelöst hat, *eventArgs* als Ausgang für Parameter, die mit dem Ereignis aufgefangen werden, und der Ausgang *triggerOut* für die Einbindung in einen Kontrollfluss.

Für die Bindung des Ereignisses und das Starten der Überwachung definiert `AbstractEvent` die Methode `void UpdateEventBinding(Object, EventInfo)`. Als ersten Parameter erwartet diese Methode einen Objektkontext bei objektgebundenen Ereignissen oder `null` bei statischen Ereignissen. Als zweiten Parameter wird die Adresse des zuletzt gebundenen Ereignisses erwartet. Die Methode wird, ebenso wie `InitPorts()`, im Setter der Eigenschaft `Event` aufgerufen, um die Bindung und Überwachung des Ereignisses einzuleiten. Die Aufgabe dieser Methode besteht aus zwei Schritten:

1. Falls die Überwachung eines Ereignisses läuft, muss diese beendet werden
2. Falls eine Adresse für ein neues Ereignis zugewiesen und, für objektgebundene Ereignisse, ein Objektkontext ungleich `null` übergeben wurde, muss die Überwachung des neuen Ereignisses gestartet werden.

Beim Starten einer Überwachung von einem Ereignis wird ein Delegat für die Methode `void Handler(Object, EventArgs)` bei dem Ereignis registriert. Beim Stoppen einer Überwachung wird der Delegat für `Handler()` bei dem Ereignis wieder deregistriert. Wird ein Ereignis ausgelöst, wird die Methode `Handler()` im Ausführungspfad (*Thread*) des Ereignisauslösers aufgerufen. Da die Ereignisse i. d. R. außerhalb der Aktivierungszeit des Brückenknotens, und damit in einem für die Laufzeitumgebung von DynamicNodes fremden Ausführungspfad, ausgelöst werden, können die beim Auffangen des Ereignisses erhaltenen Daten nicht direkt über die Ausgänge weitergeleitet werden. Zur Entkopplung der Ausführungspfade von Ereignisauslöser und Brückenknoten wird eine Warteschlange verwendet. Die Methode `Handler()` nimmt das auslösende Objekt und die Ereignisparameter entgegen und fügt diese in die Warteschlange ein. Anschließend benachrichtigt sie, mit einem Aufruf von `void INode.Activate()`[1], die Laufzeitumgebung von DynamicNodes, dass der Brückenknoten aktiviert werden soll.

`AbstractEvent` implementiert die Methode `InitPorts()` und überschreibt die Methode `Work()`. Die Methode `InitPorts()` hat lediglich die Aufgabe die Kompatibilität des Ausgangs *eventArgs* an das Ereignis anzupassen. Die Kompatibilität der Ausgänge *sender* und *triggerOut* ist

[1] `http://dynamicnodes.mastersign.de/docs/api/html/M_DynamicNode_Core_INode_Activate.htm`

konstant und wurde bereits in `Init()` festgelegt. Die Implementierung der Methode `Work()` überprüft als erstes die Warteschlange. Ist kein Eintrag in der Warteschlange, werden leere Marken über die Ausgänge *sender* und *eventArgs* weitergegeben und die Ausführung wird abgebrochen. Findet sie jedoch mindestens einen Eintrag in der Warteschlange, wird dieser entnommen und seine Werte, Auslöser des Ereignisses und Ereignisparameter, über die Ausgänge *sen*der und *eventArgs* weitergegeben. Zusätzlich wird über den Ausgang *triggerOut* eine Steuermarke weitergegeben. Zum Abschluss wird überprüft, ob die Warteschlange weitere Einträge enthält. Ist das der Fall, wird von der Laufzeitumgebung mit dem Setzen eines Flags (`Node.Reactivate`[1] `= true`) eine erneute Aktivierung angefordert, um so die weiteren Einträge in der Warteschlange iterativ abzuarbeiten.

Zur Anpassung der Visualisierung implementiert `AbstractEvent` die Methoden `GetTextLine1()`, `GetTextLine2()` und überschreibt die Methode `GetInteractionSymbol1()`. `GetTextLine1()` ermittelt mit `EventInfo.ReflectedType.Namespace` den Namensraum der Klasse, für die das aktuell gebundene Ereignis definiert ist, und gibt ihn zurück. `GetTextLine2()` ermittelt den Namen der Klasse, für die das aktuell gebundene Ereignis definiert ist, und den Namen des gebundenen Ereignisses und gibt beides mit einem Punkt als Verbindungszeichen zurück. Falls kein Ereignis gebunden ist, geben beide `GetTextLine`-Methoden `null` zurück. Die Methode `GetInteractionSymbol1()` gibt das Symbol für ein Ereignis zurück.

Die Methoden `OnStore()` und `OnRestore()` werden nach dem gleichen Muster wie in der Basisklasse `AbstractMethod` überschrieben (vgl. *AbstractMethod* auf Seite 91). Auch die Bereitstellung der Knotensteuerung, für die Auswahl der Adresse des zu bindenden Ereignisses, folgt genau dem Muster in `AbstractMethod`, wobei als Knotensteuerung das Steuerelement `EventSelector`[2] zum Einsatz kommt.

NEvent

Die Klasse `DynamicNode.Lib.Objects.NEvent` (vgl. *Listing A.27* im Anhang auf Seite 201) erbt von `AbstractEvent` und spezialisiert sie für das Überwachen von objektgebundenen Ereignissen. Sie implementiert die Methode `AbstractEvent.CheckEventInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf ein objektgebundenes Ereignis verweist.

Die Initialisierungsphase des Brückenknotens, `Init()`, wird in diesem Brückenknoten ergänzt, indem die Methode überschrieben wird, zu Beginn der Methode aber die Implementierung in der Basisklasse aufgerufen wird. Dadurch werden zunächst alle Anschlüsse wie in

[1] `http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Core_Node_Reactivate.htm`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_FieldSelector.htm`

`AbstractEvent` erzeugt. Anschließend wird ein Eingang *instance* für den Objektkontext des objektgebundenen Ereignisses erzeugt.

Auch die Implementierung der Methoden `InitPorts()` und `Work()` aus der Basisklasse `AbstractEvent` werden zwar überschrieben, dabei jedoch nicht ersetzt sondern nur ergänzt. Die Methode `InitPorts()` wird um die Anpassung der Kompatibilität des Eingangs *instance* ergänzt. Dabei wird die Kompatibilität von *instance* auf die Klasse gesetzt, für die das aktuell gebundene Ereignis definiert wurde. Die Methode `Work()` wird ergänzt um das Abfragen des Eingangs *instance*. Findet `Work()` auf dem Eingang *instance* einen neuen Objektkontext, so wird `UpdateEventBinding()` mit dem gefundenen Objektkontext aufgerufen und dadurch das aktuell adressierte Ereignis in dem neuen Objektkontext überwacht.

Zur Konfiguration der Knotensteuerung implementiert `NEvent` die Methode `AbstractEvent.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur objektgebundene Ereignisse zur Auswahl stellt.

In *Abbildung 5.11* ist die Visualisierung von einer `NEvent`-Instanz dargestellt.

Abbildung 5.12: Visualisierung des Brückenknotens `NEvent`

NStaticEvent

Die Klasse `DynamicNode.Lib.Objects.NStaticEvent` (vgl. *Listing A.28* im Anhang auf Seite 202) erbt von `AbstractEvent` und spezialisiert sie für das Überwachen von statischen Ereignissen. Sie implementiert die Methode `AbstractEvent.CheckEventInfo()` und wirft eine Ausnahme, falls die übergebene Adresse nicht auf ein statisches Ereignis verweist.

Da die Basisklasse `AbstractEvent` bereits die gesamte Funktionalität zum Überwachen von statischen Ereignissen implementiert, müssen in dieser Klasse lediglich die Visualisierung und die Konfiguration der Knotensteuerung angepasst werden. Zur Anpassung der Visualisierung überschreibt `NStaticEvent` die Methode `GetInteractionSymbol1()` und gibt das Symbol für ein statisches Ereignis zurück. Zur Konfiguration der Knotensteuerung implementiert `NStaticEvent` die Methode `AbstractEvent.InitSelector()`. Darin wird die übergebene Knotensteuerung so konfiguriert, dass sie nur statische Ereignisse zur Auswahl stellt.

In 5.13 ist die Visualisierung von einer `NStaticEvent`-Instanz dargestellt.

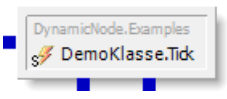


Abbildung 5.13: Visualisierung des Brückenknotens `NStaticEvent`

5.3 Steuerelemente für die Adressauswahl

Die Brückenknoten sollen Adressauswahlsteuerelemente anbieten, welche die Auswahl von Interaktionen ermöglichen (vgl. 4.3.5). Ein Benutzersteuerelement wird durch eine Klasse implementiert, die von `System.Windows.Forms.UserControl`[1] erbt. Für die verschiedenen Interaktionsformen sind unterschiedliche Adressauswahlsteuerelemente notwendig. Deshalb müssen mehrere Klassen implementiert werden, die von `UserControl` erben.

Wie in 4.3.5 beschrieben, besteht das Adressauswahlsteuerelement für einen Brückenknoten aus zwei Teilen: der Adressauwahl für die Klasse und der Adressauswahl für das Klassenmitglied. In dem folgenden Unterabschnitt wird zunächst die Adressauswahl für eine Klasse, als gemeinsamer Teil für alle Adressauswahlsteuerelemente, behandelt. Anschließend wird auf die Adressauswahl der Klassenmitglieder eingegangen. Alle Steuerelemente sind in der Assembly `DNExtObjects` und im Namensraum `DynamicNode.Ext.Object`[2] definiert.

5.3.1 Adressauswahlsteuerelement für Klassen

Jede Klasse lässt sich durch ihren voll qualifizierten Namen beschreiben. Innerhalb des .NET-Frameworks besteht dieser aus dem Namensraum, dem Klassennamen und dem Namen des Assemblies, in dem die Klasse definiert wird. Der Name eines Assemblies setzt sich wiederum aus dem Dateinamen und optional der Versionsnummer des Assemblies sowie einem Hash der Signatur (dem Public Key Token) zusammen. Da in einen Prozess, in diesem Fall den Prozess der Entwicklungsumgebung von DynamicNodes, von jeder Assembly nur eine einzige Version geladen werden kann, wird die Versionsnummer und auch das Public Key Token der Einfachheit halber für die Benennung einer Assembly weggelassen.

Die Adresse einer Klasse ist hierarchisch aufgebaut. An der Wurzel steht das Assembly, es folgt als Zweig der Namensraum und als Blatt der Klassenname. Die Auswahl einer Klasse aus allen Klassen, die in den Prozess von DynamicNodes geladen sind, lässt sich demzufolge in drei Schritten durchführen. Der erste Auswahlschritt stellt alle Assemblies zur Verfügung, die in den Prozess geladen sind. Der zweite Schritt bietet alle Namensräume zur Auswahl an, die in diesem Assembly definiert werden. Und der Dritte stellt alle Klassen zur Auswahl, die sich in dem ausgwählten Assembly und dem ausgewählten Namensraum befinden. Durch diese drei Schritte wird der Benutzer nicht mit einer unübersichtlich langen Liste konfrontiert.

Die Auswahl für jeden Schritt lässt sich platzsparend mit einer Drop-Down-Liste realisieren. Da sich Benutzersteuerelemente leicht verschachteln lassen, bietet es sich an, ein separates Benutzersteuerelement für die Auswahl einer Klasse zu implementieren. Dieses Steuerelement kann in den verschiedenen Adressauswahlsteuerelementen für die Klassenmitglieder

[1] `http://msdn.microsoft.com/de-de/library/system.windows.forms.usercontrol.aspx`
[2] `http://dynamicnodes.mastersign.de/docs/api/html/N_DynamicNode_Ext_Objects.htm`

wiederverwendet werden. Das Benutzersteuerelement für die Klassenauswahl wird `DynamicNode.Ext.Objects.TypeSelector`[1] genannt.

5.3.2 Adressauswahlsteuerelement für Klassenmitglieder

Für die Auswahl der Klassenmitglieder werden fünf verschiedene Benutzersteuerelemente implementiert. Jeweils eines für die Auswahl von Methoden, Konstruktoren, Feldern, Eigenschaften und Ereignissen. Jedes dieser Steuerelement besitzt den gleichen Aufbau. Sie verwenden das Steuerelement zur Auswahl einer Klasse wieder (vgl. 5.3.1) und bieten, ebenfalls mit einer Drop-Down-Liste, die Auswahl eines Klassenmitglieds in der jeweils ausgewählten Klasse an. Einige dieser Steuerelemente lassen sich parametrisieren.

Es gibt das Steuerelement `DynamicNode.Ext.Objects.MethodSelector`[2] Es lässt sich mit der Eigenschaft `bool MustBeStatic { get; set; }`[3] so parametrisieren, dass entweder nur statische Eigenschaften oder nur objektgebundene Eigenschaften zur Auswahl angeboten werden. Wird durch den Benutzer eine Eigenschaft ausgewählt, löst das Steuerelement das Ereignis `MethodSelector.MethodSelected`[4] aus. Die ausgewählte Methode lässt sich über die Eigenschaft `MethodInfo Method { get; set; }`[5] abrufen.

In 5.14 ist das Steuerelement `MethodSelector` in der Ansicht „Knotensteuerung" zu sehen. Das eingebettete Steuerelement `TypeSelector` ist rot umrandet und das gesamte Steuerelement `MethodSelector` ist grün umrandet.

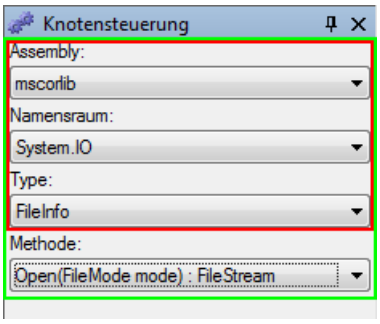


Abbildung 5.14: Adressauswahlsteuerelement für Methoden

Das nächste Steuerelement ist `DynamicNode.Ext.Objects.ConstructorSelector`[6]. Dieses

[1] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_TypeSelector.htm
[2] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_MethodSelector.htm.
[3] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_MethodSelector_MustBeStatic.htm
[4] http://dynamicnodes.mastersign.de/docs/api/html/E_DynamicNode_Ext_Objects_MethodSelector_MethodSelected.htm
[5] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_MethodSelector_Method.htm
[6] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_ConstructorSelector.htm

Steuerelement lässt sich nicht parametrisieren, da es nur eine Sorte von Konstruktoren gibt. Ähnlich wie in `MethodSelector` gibt es ein Ereignis `ConstructorSelected`[1] und eine Eigenschaft `ConstructorInfo Constructor { get; set; }`[2].

Das Steuerelement `DynamicNode.Ext.Objects.FieldSelector`[3] dient der Auswahl von Feldern. Dieses Steuerelement lässt sich durch zwei Parameter konfigurieren. Einerseits mit `bool MustBeStatic { get; set; }`[4] für die Eingrenzung der Auswahl auf statische oder objektgebundene Felder und andererseits mit `bool Writeable { get; set; }`[5] für die Eingrenzung der Auswahl auf Felder, die geschrieben werden können. Ähnlich wie in `MethodSelector` gibt es ein Ereignis `FieldSelected`[6] und eine Eigenschaft `FieldInfo Field { get; set; }`[7].

Das Steuerelement `DynamicNode.Ext.Objects.PropertySelector`[8] dient der Auswahl von Eigenschaften. Diese Steuerelement lässt sich durch zwei Parameter konfigurieren. Mit `bool MustBeStatic { get; set; }`[9] kann festgelegt werden ob statische oder objektgebundene Eigenschaften ausgewählt werden können. Mit `bool Writeable { get; set; }`[10] lässt sich festlegen ob die Auswahl auf Eigenschaften eingegrenzt wird, die lesbar sind oder ob die Auswahl auf Eigenschaften eingegrenzt wird, die beschrieben werden können. Ähnlich wie in `MethodSelector` gibt es ein Ereignis `PropertySelected`[11] und eine Eigenschaft `PropertyInfo Property { get; set; }`[12].

Das letzte Steuerelement ist `DynamicNode.Ext.Objects.EventSelector`[13]. Es dient der Auswahl von Ereignissen und lässt sich mit `bool ShowStatic-Events { get; set; }`[14] so kon-

[1] http://dynamicnodes.mastersign.de/docs/api/html/E_DynamicNode_Ext_Objects_ConstructorSelector_ConstructorSelected.htm
[2] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_ConstructorSelector_Constructor.htm
[3] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_FieldSelector.htm
[4] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_FieldSelector_MustBeStatic.htm
[5] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_FieldSelector_Writeable.htm
[6] http://dynamicnodes.mastersign.de/docs/api/html/E_DynamicNode_Ext_Objects_FieldSelector_FieldSelected.htm
[7] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_FieldSelector_Field.htm
[8] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_PropertySelector.htm
[9] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_PropertySelector_MustBeStatic.htm
[10] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_PropertySelector_Writeable.htm
[11] http://dynamicnodes.mastersign.de/docs/api/html/E_DynamicNode_Ext_Objects_PropertySelector_PropertySelected.htm
[12] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_PropertySelector_Property.htm
[13] http://dynamicnodes.mastersign.de/docs/api/html/T_DynamicNode_Ext_Objects_EventSelector.htm
[14] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_EventSelector_

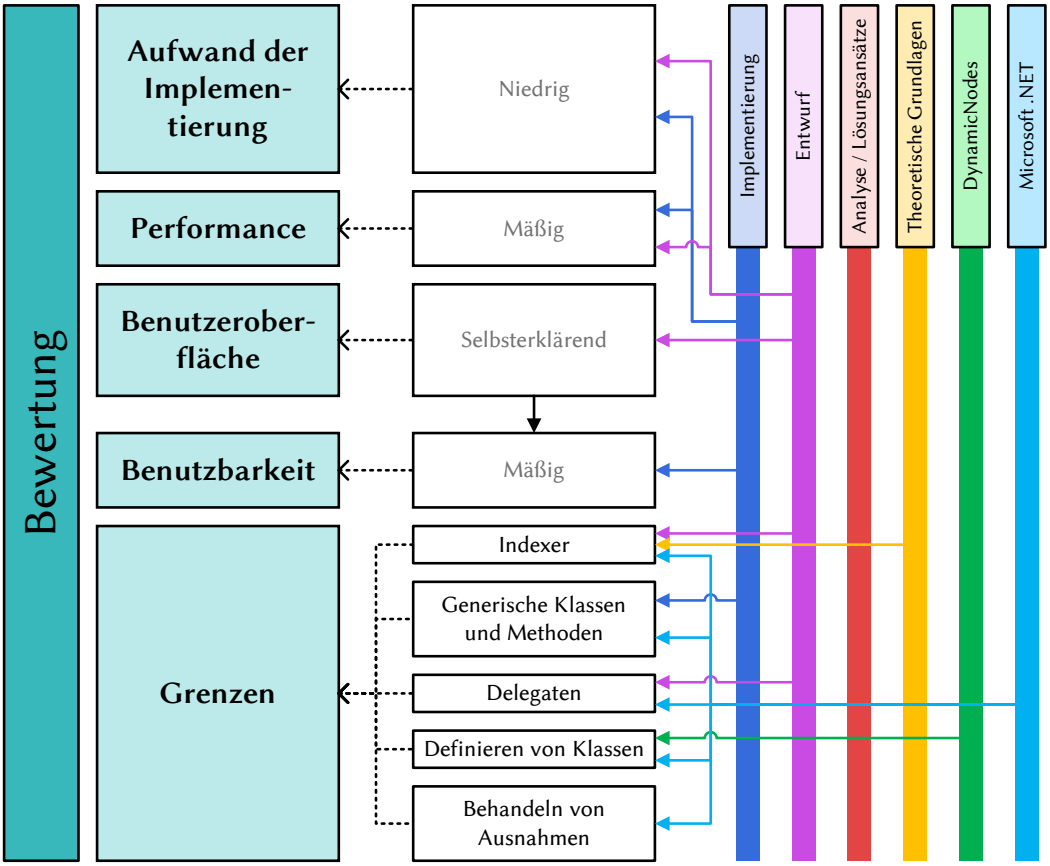
figurieren, dass entweder nur statische Ereignisse oder nur objektgebundene Ereignisse zur Auswahl gestellt werden. So wie `MethodSelector` besitzt es ein Ereignis `EventSelected`[1] und eine Eigenschaft `EventInfo Event { get; set; }`[2].

ShowStaticEvents.htm

[1] http://dynamicnodes.mastersign.de/docs/api/html/E_DynamicNode_Ext_Objects_EventSelector_EventSelected.htm

[2] http://dynamicnodes.mastersign.de/docs/api/html/P_DynamicNode_Ext_Objects_EventSelector_Event.htm

6 Bewertung


Bewertung
Aufwand der Implementierung
Performance
Benutzeroberfläche
Benutzbarkeit
Grenzen
Niedrig
Mäßig
Selbsterklärend
Mäßig
Indexer
Generische Klassen und Methoden
Delegaten
Definieren von Klassen
Behandeln von Ausnahmen
Implementierung
Entwurf
Analyse / Lösungsansätze
Theoretische Grundlagen
DynamicNodes
Microsoft .NET

Dieses Kapitel schließt den praktischen Teil der Arbeit ab und überprüft die einzelnen in *Unterabschnitt 4.1.3 Entwurfsziele* gesteckten Ziele. Anschließend werden einige Grenzen der Implementierung beschrieben, die während des Entwurfs nicht sichtbar geworden sind.

Das Kapitel gliedert in fünf Abschnitte. Die ersten vier Abschnitte behandeln die Entwurfsziele: *Abschnitt 6.1 Niedriger Implementierungsaufwand*, *Abschnitt 6.2 Hohe Performance*, *Abschnitt 6.3 Selbsterklärende Benutzeroberfläche* und *Abschnitt 6.4 Einfache Benutzbarkeit*. Im letzten Abschnitt – Abschnitt 6.5 – werden die Grenzen der Implementierung beschrieben.

6.1 Niedriger Implementierungsaufwand

Das erste Entwurfsziel, welches an dieser Stelle überprüft werden muss, ist ein niedriger Implementierungsaufwand. Da es an Vergleichsmöglichkeiten mangelt, kann die Bewertung des Implementierungsaufwandes weder relativ, noch absolut durchgeführt werden. Statt dessen kann bewertet werden, ob alle Möglichkeiten genutzt wurden, um den Implementierungsaufwand für die Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken zu senken. Dazu werden die einzelnen Entwurfsentscheidungen untersucht.

Die erste Enwurfsentscheidung ist die Auswahl eines Lösungsansatzes (vgl. *Unterabschnitt 4.3.1 Auswahl des Lösungsansatzes*). Diese Entscheidung wurde für den Lösungsansatz der polymorphen Brückenknoten getroffen. Die manuelle Implementierung der Brückenknoten, die zweifelsfrei den größten Implementierungsaufwand bedeuten würde, kommt für eine allgemeine Anbindung von allen .NET-Klassenbibliotheken nicht in Frage. Ob der Implementierungsaufwand für einen Quellcode-Generator höher wäre als die Implementierung der polymorphen Brückenknoten, lässt sich nicht ohne Weiteres abschätzen. Die Vorteile der polymorphen Knoten (kein erneutes Kompilieren erforderlich, geringe Anzahl an Knotentypen) würden jedoch auch einen eventuellen Mehraufwand rechtfertigen.

Die zweite Entwurfentscheidung ist die Wahl der Bindungstechnik (vgl. *Unterabschnitt 4.3.2 Bindung der Interaktionen*). Es wurde die lose Bindung mit Hilfe der Reflection-API gewählt, welche weniger Implementierungsaufwand bedeutet als eine feste Bindung mit Erzeugung von IL-Code zur Laufzeit. Damit wurde diese Entscheidung zugunsten des Implementierungsaufwandes getroffen.

Die dritte Entwurfentscheidung ist die Umsetzung der Polymorphie in den Brückenknoten (vgl. *Unterabschnitt 4.3.4 Polymorphie der Brückenknoten*). Wenig Aufwand ruft die Implementierung von schwach polymorphen Knotentypen hervor. Ein größerer Aufwand wird durch die Implementierung von voll polymorphen Knotentypen hervorgerufen. Durch den Entwurf werden die voll polymorphen Brückenknoten identifiziert: Die Brückenknoten für die Einbindung von statischen und objektgebundenen Methoden sowie der Brückenknoten für die Einbindung von Konstruktoren. Der Entwurf schreibt zwei kompakte Algorithmen für

die Initialisierungsphase von schwach und voll polymorphen Brückenknoten vor. Mit dieser, für die verschiedenen Brückenknoten wiederverwendbaren Vorlage, wird der Implementierungsaufwand gesenkt.

Die vierte Entwurfsentscheidung betrifft die Einrichtung von Anschlüssen für die Einbettung der Brückenknoten in einen Kontrollfluss (vgl. *Unterabschnitt 4.3.3 Anschlüsse für die Einbettung in einen Kontrollfluss*). Diese Entwurfsentscheidung lässt kaum Spielraum für eine, im Sinne des Implementierungsaufwandes, günstige oder ungünstige Lösung. Aus diesem Grund wird diese Entwurfsentschiedung nicht näher betrachtet.

Die fünfte Entwurfsentscheidung betrifft visuelle Gestaltung der Brückenknoten (vgl. *Unterabschnitt 4.3.6 Visuelle Gestaltung der Brückenknoten*). Dabei wird ein Layout für die Visualisierung der Knoten erarbeitet. Es wurde die Entscheidung getroffen, dass die Standardvisualisierung der Knoten durch eine knotenspezifische Visualisierung ersetzt werden soll. Diese Entscheidung bedeutet einen erheblichen Aufwand für die Implementierung, bringt jedoch auch einen bedeutenden Mehrwert für die Benutzerschnittstelle mit sich. Im Rahmen der Möglichkeiten für die Gestaltung der Visualisierung wurde eine verhältnismäßig einfache Darstellung gewählt, um den Aufwand zu begrenzen.

Die sechste Entwurfsentscheidung ist die Gestaltung der Klassenhierarchie für die Implementierung der Brückenknoten (vgl. *Unterabschnitt 4.3.7 Klassenhierarchie der Brückenknoten*). Diese Entscheidung wirkt sich stark auf den Implementierungsaufwand aus. Die Klassenhierarchie wurde derart entworfen, dass jene Gemeinsamkeiten der Brückenknoten, die einen hohen Implementierungsaufwand erfordern, in einer gemeinsamen Basisklasse zusammengeführt werden. Das beste Beispiel dafür ist die Implementierung der voll polymorphen Initialisierungsphase für die Knoten `NMethod`, `NStaticMethod` und `NConstructor`. In der Basisklasse `AbstractMethodBase` ist die polymorphe Initialisierungsphase nur ein einziges mal implementiert und wird von allen drei Brückenknoten wiederverwendet. Es lässt sich feststellen, dass die Struktur der Klassenhierarchie für die Brückenknoten den Implementierungsaufwand stark vermindert.

Bis auf die visuelle Gestaltung der Brückenknoten wurden alle wesentlichen Entwurfsentscheidungen zugunsten des Implementierungsaufwandes oder zumindest aus der Sicht des Implementierungsaufwandes neutral getroffen.

6.2 Hohe Performance

Das zweite Entwurfsziel ist eine hohe Performance. Um das Erreichen dieses Ziels zu bewerten, soll eine Vergleichsmessung durchgeführt werden. Dazu werden zwei Hilfsknoten manuell implementiert. Der eine Knoten steuert die Performance-Messung und misst die verbrauchte Zeit, der zweite Knoten bildet die Referenz gegenüber dem Brückenknoten. Als

Brückenknoten wird der Knoten `NStaticMethode` ausgewählt. Da alle Brückenknoten einen sehr ähnlichen Aufbau in der Arbeitsmethode `Work()` besitzen, soll an dieser Stelle die Messung mit diesem einen Brückenknoten ausreichen.

Der Versuchsaufbau sieht dazu wie folgt aus: Der Steuerknoten besitzt einen Zähler und einen Zeitmesser. Er sendet über einen Ausgang einen beliebigen Wert an den Operationsknoten, dessen Performance gemessen werden soll. Der versendete Wert muss – dem Datentyp nach – mit dem Operationsknoten kompatibel sein. In dieser Messung werden Integer-Werte verwendet. Der Operationsknoten nimmt den Wert entgegen und führt die Operation aus. Die Operation für diese Messung soll nach Möglichkeit sehr klein sein, damit sich die verbrauchte Zeit des Overheads stark in der Zeitmessung niederschlägt. Als Operation wird eine Verdopplung des Wertes mit Hilfe einer Addition durchgeführt. Das Ergebnis der Operation wird wiederum in den Steuerknoten geleitet. Dieser inkrementiert seinen internen Zähler. Ist ein Abbruchwert erreicht, stoppt er die Zeitmessung und gibt die gemessene Zeitspanne aus. Ist der Abbruchwert noch nicht erreicht, sendet er erneut eine Ganzzahl an den Operationsknoten.

Die Messung wurde mit einem Abbruchwert von 50.000 durchgeführt. Das bedeutet, es wurden 50.000 Aktivierungen des Operationsknotens durchgeführt. In der ersten Messung wurde als Operationsknoten der manuell implementierte Knoten verwendet. Dieser enthält eine öffentliche, statische Methode, welche die Addition durchführt. In der zweiten Messung wurde als Operationsknoten der Brückenknoten `NStaticMethod` verwendet, um genau dieselbe statische Methode aus dem Referenzknoten für die Addition aufzurufen.

Der Versuchsaufbau ist in *Abbildung 6.1* dargestellt. Der Steuerknoten ist in der Klasse `DynamicNode.Examples.NBench` implementiert. Der Referenzknoten ist in der Klasse `DynamicNode.Examples.NBenchOp` und seine Operation in der statischen Methode `int NBenchOp.Operation(int, int)` implementiert.

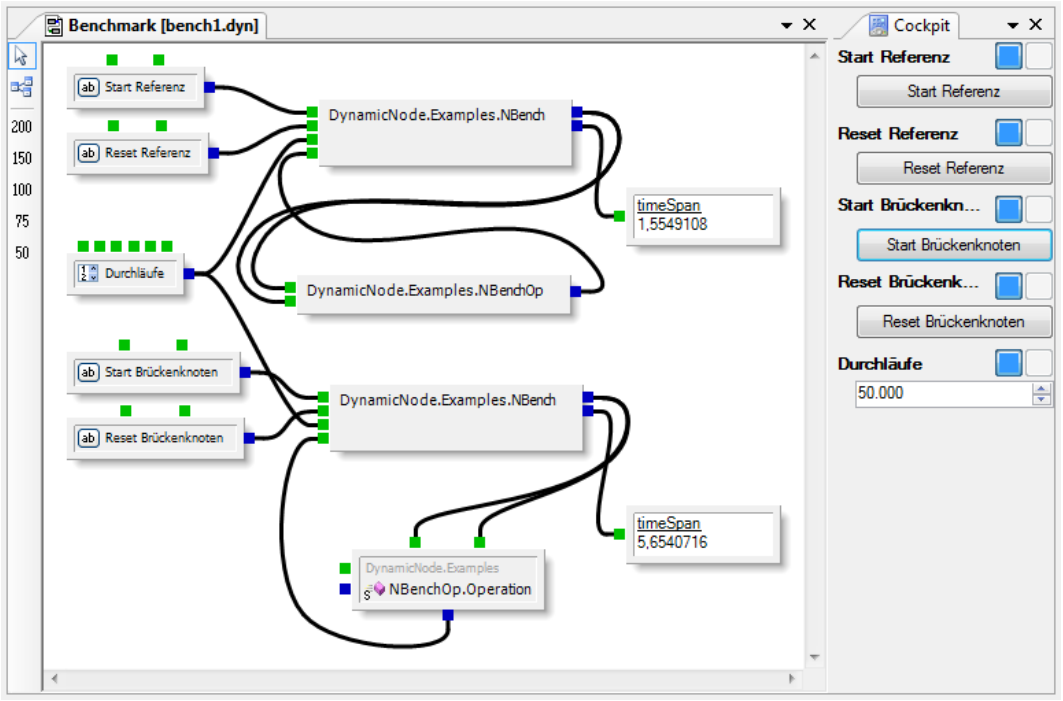


Abbildung 6.1: Aufbau für die Performance-Messung eines Brückenknotens

Beide Messungen wurden mit dem DynamicNodes-Kern in der Version 1.4.3777.29483 durchgeführt. Das Testsystem besitzt einen Athlon 64 X2 4600+ und 3 GB Arbeitsspeicher. Als Betriebssystem kam Windows 7 in der 64bit-Variante zum Einsatz. DynamicNodes wurde im 32bit-Modus ausgeführt.

Die folgenden Zeitspannen wurden gemessen:

- Messung mit manuell implementiertem Operationsknoten: 1,56 Sekunden
- Messung mit `NStaticMethod`: 5,65 Sekunden

Diese Ergebnisse zeigen, dass der Brückenknoten für den Aufruf einer statischen Methode einen mindestens dreimal so großen Overhead erzeugt, wie der Aufruf durch einen nativ implementierten Knoten. Der größte Anteil dieses Overhead ist auf die Nutzung der Reflection-API zurückzuführen (vgl. *Unterabschnitt 4.3.2 Bindung der Interaktionen*). Da die Nutzung der Reflection-API eine einfache Implementierung ermöglicht hat, ist der dreifache Overhead für eine Beispielimplementierung im Rahmen dieser Arbeit ein gutes Ergebnis. Durch eine aufwändigere Implementierung mit fester Bindung könnte die Performance jedoch deutlich gesteigert werden.

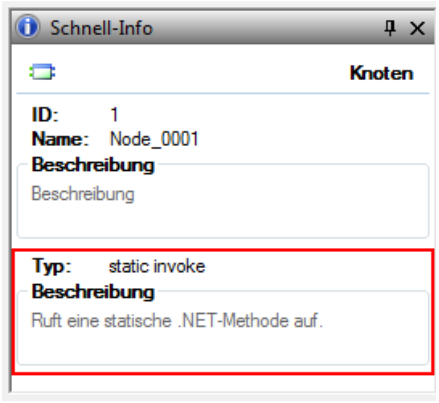


Abbildung 6.2: Der Beschreibungstext eines Brückenknotens in der Ansicht „Schnell-Info"

6.3 Selbsterklärende Benutzeroberfläche

Das dritte Entwurfziel ist eine selbsterklärende Benutzeroberfläche. Damit beurteilt werden kann, ob dieses Ziel erreicht wurde, müssen die Entscheidungen überprüft werden, die für die Gestaltung der Benutzeroberfläche getroffen wurden. Die Benutzeroberfläche eines Brückenknotens besteht einerseits aus der Visualisierung auf der Arbeitsfläche und andererseits aus seiner Knotensteuerung. Der Entwurf der visuellen Gestaltung wird in *Unterabschnitt 4.3.6 Visuelle Gestaltung der Brückenknoten* beschrieben.

Jeder Knoten definiert einen kurzen Beschreibungstext, welcher die Aufgabe des Knotens erläutert. Dieser Beschreibungstext wird in der Ansicht „Schnell-Info" eingeblendet, sobald der Benutzer den Mauszeiger über einen Knoten bewegt (vgl. *Abbildung 6.2*). Dass alle Brückenknoten einen solchen Beschreibungstext definieren, trägt dazu bei, dass ein Benutzer sich die Benutzung ohne weitere Dokumentation erschließen kann.

Die erste Entscheidung, die während dem Entwurf getroffen wird, ist ob und wie die Brückenknoten ihre eigene Visualisierung selbst zeichnen sollen. Diese Entscheidung wird positiv entschieden. Die Brückenknoten stellen in der knotenspezifischen Visualisierung die folgenden Informationen übersichtlich dar: Namensraum und den Namen der Klasse für welche die gebundene Interaktion definiert ist, den Namen Interaktion (bzw. des Klassenmitglieds) und ein oder zwei Symbole, die aus der Symbolsprache von Microsoft Visual Studio entnommen sind.

Für einen Benutzer, der mit dem .NET-Framework vertraut ist – was an dieser Stelle vorausgesetzt wird – sind der Namensraum, der Name der Klasse und der Name des Klassenmitglieds selbsterklärend und beschreiben präzise die gebundene Interaktion. Auch die Symbole, welche die Interaktionsform visualisieren, sind dem Benutzer i. d. R. vertraut und helfen dabei die konkrete Tätigkeit eines Brückenknotens in einem Datenflussgraphen zu erkennen.

Die Gestaltung der Visualisierung der Brückenknoten trägt wesentlich zu einer selbsterklä-

renden Benutzeroberfläche bei.

Die nächste Entwurfsentscheidung ist, ob der Name des Knotentyps angezeigt werden soll. Der Name eines Knotentyps wird durch die Darstellung der Interaktionsform mit Hilfe von Symbolen zu einer redundanten Information und raubt Platz auf der Arbeitsfläche. Es wurde entschieden, dass der Name des Knotentyps ausgeblendet werden soll. Diese Entscheidung trägt, wenn auch nicht zu einer selbsterklärenden Benutzeroberfläche, dann doch zumindest zu einer übersichtlichen Oberfläche bei.

Die letzte Entwurfsentscheidung für die grafische Darstellung der Brückenknoten ist die Anordnung der Anschlüsse. Der Entwurf schreibt ein einheitliches Schema für die Anordnung der Anschlüsse vor, welches von allen Brückenknoten durch ihre Implementierung umgesetzt wird. Dabei werden die Anschlüsse für die Einbettung in einen Kontrollfluss an der linken Kante angeordnet und so optisch von den übrigen Ein- und Ausgängen abgesetzt. Alle Eingänge für Parameter werden an der Oberkante der Brückenknoten und die Ausgänge für Ergebnisdaten an der Unterkante der Brückenknoten angeordnet. Diese Trennung folgt einem Datenfluss von oben nach unten. Die Anordnung der Anschlüsse ist übersichtlich und trägt zu einer selbsterklärenden Benutzeroberfläche bei.

Die Anschlüsse, welche durch die Polymorphie einiger Brückenknoten erzeugt werden, erhalten Namen, die mit den Parameternamen der gebundenen Interaktion übereinstimmen. So erkennt ein Programmierer die aus der .NET-Programmierung bekannten Methoden- oder Konstruktorparameter in den Anschlüssen der Knoten wieder. Der Name eines Anschlusses wird von DynamicNodes in der Ansicht „Schnell-Info" eingeblendet, sobald der Mauszeiger über einen Anschluss bewegt wird.

Um abschließend bewerten zu können, ob die Benutzeroberfläche der Brückenknoten selbsterklärend gestaltet wurde, muss auch die Gestaltung der Knotensteuerung beurteilt werden. Die Knotensteuerungen dienen der Auswahl einer konkreten Interaktionsform. Der Arbeitsablauf für einen Benutzer der Brückenknoten sieht wie folgt aus: Er zieht aus der Knotenbibliothek einen Brückenknoten auf die Arbeitsfläche und wählt ihn aus. Daraufhin wird die Knotensteuerung des Brückenknotens angezeigt und der Benutzer kann eine Interaktion auswählen. Ist die Auswahl vollzogen, passt der Brückenknoten bei Bedarf seine Anschlüsse an. Anschließen kann der Brückenknoten durch den Aufbau von neuen Verbindungen in den Datenfluss eingebunden werden.

Die Knotensteuerung zerlegt die Auswahl eines Klassenmitglieds in der Hierarchie aller geladenen Klassen und Klassenmitglieder in mehrere Stufen. Zunächst hat der Benutzer die Möglichkeit ein Assembly auszuwählen. Anschließend werden ihm alle Namensräume innerhalb des gewählten Assemblies zur Auswahl gestellt. Ist ein Namensraum gewählt, kann der Benutzer aus allen Klassen innerhalb des vorher gewählten Namensraums und innerhalb des vorher gewählten Assemblies wählen. Ist die Klasse ausgewählt, werden alle Klassenmitglie-

der der gewählten Klasse zur Auswahl angeboten. Dabei werden nur jene Klassenmitglieder angezeigt, die von dem Brückenknoten gebunden werden können (vgl. *Abbildung 5.14*).

Diese mehrstufige Auswahl zwingt dem Benutzer einerseits eine Handlungsreihe auf, andererseits führen die eingeschränkten Auswahlmöglichkeiten aber auch dazu, dass der Benutzer in jedem Schritt eine überschaubare Menge an Auswahlmöglichkeiten präsentiert bekommt und er nicht mit übermäßig langen Listen konfrontiert wird. Diese Art der Gestaltung für die Knotensteuerung ist selbsterklärend.

Zusammenfassend lässt sich sagen, dass alle Entscheidungen während des Entwurfs der Benutzeroberfläche für die Brückenknoten, die Entstehung einer selbsterklärenden Benutzeroberfläche ermöglichen und unterstützen oder zumindest der Übersichtlichkeit der Benutzeroberfläche zuträglich sind.

6.4 Einfache Benutzbarkeit

Das dritte Entwurfsziel ist einfache Benutzbarkeit. Für die Bewertung der Implementierung unter dem Gesichtspunkt dieses Kriteriums muss der Arbeitsablauf untersucht werden, der von dem Benutzer während der Nutzung der Brückenknoten gefordert wird. Wie bereits in *Abschnitt 6.3 Selbsterklärende Benutzeroberfläche* beschrieben wurde, setzt sich die Arbeit mit einem Brückenknoten aus zwei Schritten zusammen.

Als erstes muss der Benutzer entscheiden, welche Interaktionsform die konkrete Interaktion besitzt, die er in den Datenflussgraphen einbinden möchte. Denn nach der Interaktionsform muss er den passenden Brückenknoten in der Knotenbibliothek wählen. Dazu muss der Benutzer bereits genau wissen, welche konkrete Interaktion er einbinden möchte.

Als nächstes muss der Benutzer den passenden Brückenknoten aus der Knotenbibliothek auf die Arbeitsoberfläche ziehen und ihn damit instanzieren. Ist dies geschehen, muss der Benutzer den Knoten auswählen und in der Knotensteuerung vier Auswahlschritte durchführen, bis er das Klassenmitglied ausgewählt hat, mit dem der Datenflussgraph interagieren soll. Erst anschließend kann er beginnen den Brückenknoten mit Verbindungen in den Datenfluss einzubinden.

Bis zu diesem Zeitpunkt sind mindestens 11 Mausklicks notwendig (die Drag&Drop-Aktion für Instanzierung als zwei Mausklicks gezählt). Und das ist nur der günstige Fall, in dem bei allen Auswahlschritten in der Knotensteuerung der auszuwählende Eintrag sofort sichtbar ist und somit kein Scrollen notwendig wird. Bei einem gewöhnlichen manuell implementierten Knoten reicht i. d. R. die Drag&Drop-Aktion für die Erzeugung des Knotens aus. Das bedeutet, dass die Benutzung von Brückenknoten für einen Benutzer von DynamicNodes – hervorgerufen durch die Polymorphie der Brückenknoten – wesentlich aufwändiger ist, als

die Verwendung von manuell implementierten Knoten, die nach ihrer Erzeugung direkt verwendbar sind.

Diese aufwändige Handhabung lässt sich jedoch nur durch einen wesentlich erhöhten Implementierungsaufwand für die Knotensteuerungen oder durch eine Erweiterung der Entwicklungsumgebung von DynamicNodes vereinfachen. Beides wurde in dieser Arbeit bewusst vermieden.

6.5 Grenzen

Die in dieser Arbeit implementierte Brücke zwischen DynamicNodes und den .NET-Klassenbibliotheken besitzt einige Schwächen, die sich nicht aus einem Verfehlen der Entwurfziele ergeben. Die meisten dieser Schwächen wurden bewusst eingegangen, um den Rahmen der Arbeit nicht zu sprengen. In diesem Abschnitt werden diese Schwächen erläutert.

6.5.1 Eigenschaften mit Index-Parametern

In *Unterabschnitt 2.3.2 Besonderheiten der .NET-Klassen* werden die Eigenschaften als Klassenmitglieder in .NET-Klassen vorgestellt. In diesem Unterabschnitt wurde nicht erwähnt, dass es eine spezielle Form von Eigenschaften gibt, die einen oder mehrere Index-Parameter besitzen können. Mit diesen Index-Parametern wird in den .NET-Klassen der Zugriff durch einen Array-Operator ermöglicht. Die Eigenschaften werden mit dem Namen `this`-Deklariert und können nur über den Array-Operator `[]` verwendet werden. Die folgende Beispielklasse zeigt die Deklaration einer Eigenschaft mit einem Index-Parameter.

```
1  class Bauer
2  {
3          private List<string> kinder = new List<string>();
4
5          public void NeuesKind(string name)
6          {
7                  kinder.Add(name);
8          }
9
10         public int AnzahlDerKinder
11         {
12                 get { return kinder.Count; }
13         }
14
15         public string this[int index]
16         {
```

```
17                  get { return kinder[index]; }
18                  set { kinder[index] = value; }
19          }
20  }
```

Der folgende Code-Schnipsel zeigt die Verwendung einer Eigenschaft mit Index-Parameter. Der Zugriff auf die Namen der Kinder des Bauern ist mit dem Array-Operator und dem Index wie bei einem gewöhnlichen Array möglich.

```
var bauer = new Bauer();
bauer.NeuesKind("Walter");
bauer.NeuesKind("Birgit");
Console.WriteLine(bauer.AnzahlDerKinder);
Console.WriteLine(bauer[1]);
---
2
Birgit
```

Das Lesen und Schreiben von Eigenschaften mit Index-Parametern ist mit den implementierten Brückenknoten nicht möglich.

6.5.2 Auswahl der Adresse von generischen Klassen und Klassenmitgliedern

Das .NET-Framework unterstützt seit der Version 2.0 generische Klassen und Methoden. Ein gutes Beispiel für eine generische Klasse ist die Klasse `System.Collections.Generic.List<T>`[1]. Sie implementiert ein typensicheres dynamisches Array. Bei generischen Klassen wird zwischen der generischen Klassendefinition und der konkreten generischen Klasse unterschieden. Eine generische Klassendefinition könnte so aussehen:

```
1   class Behälter<T>
2   {
3           private T inhalt;
4
5           public T Inhalt
6           {
7                   get { return inhalt; }
8                   set { inhalt = value; }
9           }
10  }
```

Eine generische Klassendefinition kann nicht instanziert werden ohne die Typ-Parameter (in diesem Fall `T`) auszufüllen. Eine generischen Klasse der generischen Klassendefinition `Behäl-`

[1] `http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx`

ter<T> ist z. B. Behälter<string> oder Behälter<System.IO.FileInfo>. In dem folgenden Code-Schnipsel wird die Verwendung von generischen Klassen auf der Basis der generischen Klassendefinition Behälter<T> demonstriert.

```
var b1 = new Behälter<string>();
b1.Inhalt = "Hallo␣Welt.";
string text = b1.Inhalt;
Console.WriteLine(text);
var b2 = new Behälter<DateTime>();
b2.Inhalt = DateTime.Now;
DateTime zeit = b2.Inhalt;
Console.WriteLine(zeit.ToString());
---
Hallo Welt.
12.05.2010 10:21:06
```

Mit den implementierten Adressauswahlsteuerelementen ist es nicht möglich generische Klassen oder generische Methoden auszuwählen (vgl. *Unterabschnitt 5.3.1 Adressauswahlsteuerelement für Klassen*). Dazu müsste für jeden Typ-Parameter einer generischen Klassendefinition oder einer generischen Methodendefinition die Auswahl der Adresse einer Klasse möglich sein. Diese kann wiederum die Adresse einer generischen Klasse sein, wodurch die Auswahl der Adresse einer Klasse zu einem rekursiven Prozess wird. Der Entwurf eines Adressauswahlsteuerelements für diesen Zweck hätte sich zu weit vom Kernthema der Arbeit entfernt und wurde aus diesem Grund nicht behandelt.

6.5.3 Erzeugen von Delegaten

Einige .NET-APIs fordern die Übergabe von Delegaten an Methoden. Ein Beispiel ist der Namensraum System.Threading[1], dessen Klassen für parallele Programmierung verwendet werden. Viele dieser Klassen verwenden Delegaten, um eine Aufgabe entgegenzunehmen, die parallel ausgeführt werden soll. Der folgende Code-Schnipsel zeigt ein Beispiel.

```
1 public void ArbeiteParallel()
2 {
3         var arbeit = new ThreadStart(Aufgabe);
4         var thread = new Thread(arbeit);
5         thread.Start();
6 }
7
8 private void Aufgabe()
9 {
```

[1] http://msdn.microsoft.com/en-us/library/798axes2.aspx

```
10          Console.WriteLine("Ich␣laufe␣parallel.");
11  }
```

In der Arbeit wurde kein Brückenknoten entworfen, der einen Datenflussgraphen oder einen Teil eines Datenflussgraphen als Methode kapselt und von dieser Methode einen Delegaten erzeugt. Aus diesem Grund können Interaktionen, welche auf die Verwendung von Delegaten angewiesen sind, nicht mit den implementierten Brückenknoten an DynamicNodes angebunden werden.

6.5.4 Definieren von Klassen und Implementieren von Schnittstellen

Die Funkionen einer .NET-Klassenbibliothek liegen nicht immer als konkrete Klasse vor, die instanziert und verwendet werden kann. Vielfach entfaltet sich das Potential einer .NET-Klassenbibliothek erst, wenn Schnittstellen implementiert oder neue Klassen von abstrakten Basisklassen abgeleitet werden. Ein Beispiel ist die Methode `string Int32.ToString(IFormatProvider)`[1]. Diese Methode erzeugt aus einer Ganzzahl eine formatierte Zeichenkette. Dazu verwendet sie ein Objekt, dessen Klasse die Schnittstelle `IFormatProvider`[2] implementiert. Das .NET-Framework enthält einige Klassen, die das tun. Soll die Formatierung aber vollständig frei angepasst werden, muss eine eigene Klasse implementiert werden.

Mit den implentierten Brückenknoten ist es nicht möglich Klassen zu definieren. Folglich können auch keine Schnittstellen implementiert oder neue Klassen von Basisklassen abgeleitet werden. Ist die Funktionalität einer .NET-Klassenbibliothek nur zugänglich, wenn das Definieren von neuen Klassen möglich ist, dann kann diese .NET-Klassenbibliothek mit den implementierten Brückenknoten nicht in DynamicNodes eingebunden werden.

6.5.5 Behandlung von Ausnahmen

Das .NET-Framework verwendet ein Ausnahmensystem zur Behandlung von Fehlern, die während der Programmausführung auftreten können. Dazu gehören z. B. das Verwenden von ungültigen Indizes beim Zugriff auf Arrays oder auch der Versuch eine Datei zu öffnen, die nicht existiert. Auch Zugriffsrechte oder Konfigurationsfehler können zu solchen Ausnahmen führen.

Ausnahmen sind Objekte einer Klasse, die von `System.Exception`[3] erbt. Für den Umgang mit Ausnahmen definiert die Sprache C# die Schlüsselwörter `throw`, `try`, `catch` und `finally`. Die genaue Verwendung dieser Schlüsselwörter kann in [Rich06, S. 419ff] nachgelesen werden. Die Konsequenz des Ausnahmenmodells von .NET für die Brückenknoten ist, dass bei nahezu

[1] `http://msdn.microsoft.com/en-us/library/cht2hdff.aspx`
[2] `http://msdn.microsoft.com/en-us/library/system.iformatprovider.aspx`
[3] `http://msdn.microsoft.com/en-us/library/system.exception.aspx`

jeder Interaktion damit gerechnet werden muss, dass die Interaktion eine Ausnahme wirft, die anzeigt, dass die Ausführung der Interaktion fehlgeschlagen ist.

In der Implementierung der Brückenknoten wird dies zwar durch einen `try-catch`-Block berücksichtigt, sodass Ausnahmen innerhalb der Brückenknoten abgefangen und nicht zu einer Ausnahme im Datenflussprogrammiersystem werden. Die einzige Reaktion, die durch die Implementierung der Brückenknoten auf eine Ausnahme ausgelöst wird, ist eine Fehlernachricht im Nachrichtensystem von DynamicNodes. Damit ist es dem Datenflussprogrammierer möglich, die Ausnahme zu erkennen, es ist jedoch nicht möglich den Datenfluss so zu modellieren, dass er automatisch auf die Ausnahme reagiert. In der Regel führt das dazu, dass der Datenfluss bei dem Auftreten einer Ausnahme an dem entsprechenden Brückenknoten stoppt und die nachfolgenden Knoten nicht weiterarbeiten können.

7 Zusammenfassung und Ausblick

Das Ziel[1] der Arbeit war es, ein allgemeines Konzept für die Einbindung von imperativen, objektorientierten Klassenbibliotheken in datenflussorientierte Systeme zu entwickeln und eine konkrete Brücke zwischen dem visuellen Datenflussprogrammiersystem DynamicNodes (vgl. [Kier09a]) und den .NET-Klassenbibliotheken (vgl. [Micr10a, Micr10b]) zu entwerfen und zu implementieren.

Zusammenfassung

Die Motivation[2] für diese Aufgabenstellung entstand aus dem Bestreben des Autors, die Leistungsfähigkeit des Datenflussprogrammiersystems DynamicNodes zu verbessern (vgl. [Kier08]).

Als Grundlage für den theoretischen Teil der Arbeit dient ein Überblick der relevanten Programmierparadigmen. Dazu gehört die datenflussorientierte[3], die imperative[4], die prozedurale[5], die objektorientierte[6] und die in aktuellen Hochsprachen weitverbreitete Mischform der imperativen, objektorientierten Programmierung[7]. Als Beispiele für die genannten Programmierparadigmen werden einige Programmiersprachen[8] genannt und im Hinblick auf ihre unterstützten Programmierparadigmen verglichen.

Die theoretische Grundlage für den praktischen Teil der Arbeit ist ein Überblick über die Entstehung und Besonderheiten des Microsoft .NET-Frameworks[9]. Einige für diese Arbeit besonders wichtigen Technologien[10] des .NET-Frameworks werden dabei genauer erläutert.

Der Schwerpunkt des theoretischen Anteils wird durch *Kapitel 3 Analyse und abgeleitete Lösungsansätze* gebildet. Zu Beginn werden die Struktur und die Programmablaufkontrolle von Datenflussgraphen[11] und imperativen, objektorientierten Klassenbibliotheken[12] untersucht.

[1] *Abschnitt 1.2 Aufgabenstellung*
[2] *Abschnitt 1.1 Motivation*
[3] *Unterabschnitt 2.1.1 Datenflussorientierte Programmierung*
[4] *Unterabschnitt 2.1.2 Imperative Programmierung*
[5] *Unterabschnitt 2.1.3 Prozedurale Programmierung*
[6] *Unterabschnitt 2.1.4 Objektorientierte Programmierung*
[7] *Unterabschnitt 2.1.5 Imperative, objektorientierte Programmierung*
[8] *Abschnitt 2.2 Programmiersprachen*
[9] *Abschnitt 2.3 Microsoft .NET-Framework*
[10] *Unterabschnitt 2.3.3 Für diese Arbeit wichtige .NET-Technologien*
[11] *Unterabschnitt 3.1.1 Datenflussgraph*
[12] *Unterabschnitt 3.1.2 Imperative, objektorientierte Klassenbibliothek*

Dabei wird der Schwerpunkt der Analyse auf die Interaktionsfähigkeit der jeweiligen Programme gelegt, denn daraus lassen sich die Anforderungen für eine Brücke zwischen den Systemen ableiten.

Aufbauend auf den Analyseergebnissen werden zwei Konzepte vorgestellt. Das Konzept der Brückenknoten[1] im Allgemeinen und das Konzept der polymorphen Knotentypen[2]. Dabei wird ein Brückenknoten als ein Knoten beschrieben, der den Datenaustausch zwischen einem Datenflussgraphen und einem Fremdsystem gestattet. Dieses Konzept wird anschließend für die Einbindung von imperativen, objektorientierten Klassenbibliotheken spezialisiert[3] dargestellt. Dazu werden mögliche Interaktionen mit einer imperativen, objektorientierten Klassenbibliothek wie der Methodenaufruf, das Instanzieren eines Objektes oder das Schreiben eines Feldes berücksichtig.

Das Konzept der polymorphen Knotentypen erlaubt die Implementierung von Knotentypen, die während oder nach ihrer Inztanzierung ihre Ein- und Ausgänge dynamisch konfigurieren. Damit wird es möglich einen Knotentyp für eine ganze Klasse von Operationen zu implementieren. Die Folge ist, dass der Implementierungsaufwand für bestimmte Arten von Knotenbibliotheken gesenkt werden kann.

Beide Konzepte werden anschließend in drei Lösungsansätzen für den Entwurf einer Brücke zwischen einem datenflussorientierten Programmiersystem und einer imperativen, objektorientierten Klassenbibliothek verwendet.

Der erste Lösungsansatz schlägt die manuelle Implementierung[4] von Knotentypen für jede einzelne Interaktion vor. Die Knotentypen können bei diesem Ansatz gut für einzelne Interaktionen optimiert werden. Der Implementierungsaufwand der einzelnen Knotentypen ist gering. Der bedeutenste Nachteil dieses Lösungsansatzes ist der hohe Aufwand, wenn eine große Anzahl von unterschiedlichen Interaktionen benötigt wird. Soll z. B. eine Klassenbibliothek mit 100 Klassen, die jeweils 20 Klassenmitglieder besitzen, in ein datenflussorientiertes Programmiersystem eingebunden werden, müssen $100 * 20 = 2000$ Knotentypen implementiert werden. Das erzeugt nicht nur viel redundanten Quellcode und ist zudem schlecht zu warten, sondern erschwert auch die Verwaltung der Knotentypen in dem Programmiersystem.

Der zweite Lösungsansatz entwickelt den Ersten weiter und schlägt die Implementierung eines Code-Generators[5] vor, der auf der Basis von Metadaten über die einzubindende Klassenbibliothek den Quell-Code für die erforderlichen Knotentypen generiert. Dieser Ansatz verringert den Aufwand bei einer großen Anzahl von einzubindenden Interaktionen. Dabei

[1] *Unterabschnitt 3.2.1 Das Konzept der Brückenknoten*
[2] *Unterabschnitt 3.2.3 Das Konzept der polymorphen Knotentypen*
[3] *Unterabschnitt 3.2.2 Das Konzept der Brückenknoten für die Einbindung von imperativen Klassenbibliotheken*
[4] *Unterabschnitt 3.3.1 Manuell implementierte Brückenknoten*
[5] *Unterabschnitt 3.3.2 Automatisch generierte Brückenknoten*

verschärft sich das Problem der Verwaltung der Knotentypen im Programmiersystem aber weiter. Denn mit einem Code-Generator ist es leicht möglich eine Klassenbibliothek mit tausenden Klassen einzubinden. Daraus resultieren jedoch auch zehntausende Knotentypen. Die Implementierung des Code-Generators ist im Vergleich zur manuellen Implementierung einzelner Brückenknoten aufwändiger.

Der dritte Lösungsansatz macht von dem Konzept der polymorphen Knotentypen gebrauch. Er schlägt vor, für jede Klasse von gleichartigen Interaktionen (z. B. für alle Methodenaufrufe) einen einzigen polymorphen Knotentyp[1] zu implementieren. Dieser Ansatz setzt voraus, dass das datenflussorientierte System polymorphe Knotentypen unterstützt. Der Vorteil dieses Ansatzes besteht darin, dass mit einer kleinen konstanten Anzahl von Knotentypen eine beliebige Anzahl von imperativen, objektorientierten Klassenbibliotheken mit allen Interaktionen eingebunden werden kann. So wie der zweite Lösungsansatz setzen auch die polymorphen Knotentypen maschinenlesbare Metadaten voraus, welche die Struktur der Klassenbibliotheken beschreiben. Ein Nachteil dieses Ansatzes ist die höhere Komplexität der Implementierung im Vergleich zu manuell implementierten Knotentypen.

Der praktische Teil der Arbeit gliedert sich in drei Kapitel: „Entwurf" (S. 50), „Implementierung" (S. 85) und „Bewertung" (S. 112). Aufgabe des praktischen Teils ist es, die .NET-Klassenbibliotheken in das Datenflussprogrammiersystem DynamicNodes einzubinden. Ziele für den Entwurf[2] sind: „Niedriger Implementierungsaufwand", „Hohe Performance", „Selbsterklärende Benutzeroberfläche" und „Einfache Benutzbarkeit".

Als Grundlage für den Entwurf dient eine Darstellung des Entwurfsraums[3]. Diese Darstellung ist nicht vollständig, sondern greift lediglich die Schlüsselentscheidungen für den Entwurf heraus. Dazu gehören die Auswahl eines Lösungsansatzes, die Auswahl einer Technik zur Bindung der Interaktionen, die Form der Polymorphie, die Anschlüsse für die Einbettung in einen Kontrollfluss, die visuelle Gestaltung der Brückenknoten und die Klassenhierarchie für ihre Implementierung.

Als Lösungsansatz[4] werden polymorphe Knotentypen gewählt. Da mit den .NET-Klassenbibliotheken eine unbekannte Anzahl von Interaktionen eingebunden werden soll, scheidet eine manuelle Implementierung aus. Auch eine automatische Generierung von Knotentypen ist problematisch, da nach jeder Code-Generierung der Compiler für die Übersetzung der Knotentypen verwendet werden muss und so kein kontinuierliches Arbeiten für den Datenflussprogrammierer möglich wäre.

Für die Bindung der Interaktionen[5] steht eine lose Bindung mit Hilfe der Reflection-API und

[1] *Unterabschnitt 3.3.3 Polymorphe Brückenknoten*
[2] *Unterabschnitt 4.1.3 Entwurfsziele*
[3] *Abschnitt 4.2 Entwurfsraum*
[4] *Unterabschnitt 4.3.1 Auswahl des Lösungsansatzes*
[5] *Unterabschnitt 4.3.2 Bindung der Interaktionen*

eine feste Bindung mit Hilfe von zur Laufzeit generiertem IL-Code zur Auswahl. Die lose Bindung hat den Vorteil einer einfachen Implementierung, ist aber vergleichsweise inperformant. Die feste Bindung ist aufwändig in der Implementierung, gewährleistet jedoch eine gute Performance. Zugunsten der Anschaulichkeit wird im Rahmen dieser Arbeit die lose Bindung gewählt.

Für die Realisierung der Polymorphie[1] in den Brückenknoten müssen zwei Fragen beantwortet werden. Welche Knotentypen sollen voll polymorph und welche schwach polymorph sein? Schwach polymorphe Knotentypen können Anschlüsse neu konfigurieren, jedoch keine Anschlüsse hinzufügen oder entfernen. Voll polymorphe Knotentypen können auch neue Anschlüsse erzeugen und überflüssige entfernen. Nur drei Interaktionsformen benötigen eine dynamische Anzahl von Ein- oder Ausgängen und müssen voll polymorph sein: der statische und der objektgebundene Methodenaufruf und der Konstruktoraufruf. Alle übrigen Interaktionsformen kommen mit einer festen Anzahl von Anschlüssen aus und können somit als schwach polymorphe Knotentypen implementiert werden.

Die zweite Frage, welche die Polymorphie der Brückenknoten beeinflusst, ist: Zu welchem Zeitpunkt sollen die Brückenknoten konfiguriert werden? Es werden polymorphe Knotentypen von polymorphen Knoten unterschieden. Polymorphe Kontentypen können ihre Anschlüsse nur während der Instanzierung konfigurieren. Polymorphe Knoten können ihre Anschlüsse beliebig oft nach der Instanzierung konfigurieren. DynamicNodes bietet keine Schnittstelle für eine Konfiguration während der Instanzierung. Deshalb sind Brückenknoten in DynamicNodes polymorphe Knoten und ihre Konfiguration wird nach der Instanzierung durchgeführt.

Die nächste Entwurfsentscheidung betrifft die Gestaltung der Anschlüsse für die Einbettung der Brückenknoten in einen Kontrollfluss[2]. Bis auf die beiden Brückenknoten, welche statische und objektgebundene .NET-Ereignisse überwachen, erhalten alle Brückenknoten einen Eingang *triggerIn* für die Steuerung durch einen Kontrollfluss und einen Ausgang *triggerOut* für die Weitergabe eines Kontrollflusses. Die Schwierigkeit bei der Einbettung eines Brückenknotens in einen Kontrollfluss ist die richtige Reaktion auf einlaufende Marken. Dazu werden zwei Betriebsmodi definiert. In dem einen Betriebsmodus beginnt der Brückenknoten zu arbeiten, sobald an einem beliebigen Eingang eine neue Marke eintrifft. Im Zweiten beginnt der Brückenknoten nur dann zu arbeiten, wenn über *triggerIn* eine neue Marke einläuft. In welchem der beiden Betriebsmodi sich der Brückenknoten befindet, wird dadurch entschieden, dass überprüft wird, ob *triggerIn* mit einem Ausgang verbunden ist oder nicht.

Ein wichtiger Bereich bei dem Entwurf eines Knotens für ein visuelles Datenflussprogrammiersystem ist seine visuelle Gestaltung[3]. Das DynamicNodes-System bietet für jeden Kno-

[1] *Unterabschnitt 4.3.4 Polymorphie der Brückenknoten*
[2] *Unterabschnitt 4.3.3 Anschlüsse für die Einbettung in einen Kontrollfluss*
[3] *Unterabschnitt 4.3.6 Visuelle Gestaltung der Brückenknoten*

tentypen automatisch eine Standardvisualisierung. Da diese für eine aussagekräftige Darstellung aber nicht ausreicht, wird für die Brückenknoten eine spezialisierte grafische Darstellung implementiert. Diese besteht aus einem zweizeiligen Layout. In der ersten Zeile steht der Namensraum der Klasse, zu dem die gebundene Interaktion gehört. In der zweiten Zeile werden ein oder zwei passende Symbole, gefolgt von Klassen- und Klassenmitgliedsname der Interaktion, dargestellt.

Die letzte Entwurfsentscheidung, die ausführlich diskutiert wird, ist die Gestaltung der Klassenhierarchie[1] für die Implementierung der Brückenknoten. Dabei wird die visuelle Gestaltung als eine Fähigkeit aller Brückenknoten identifiziert und deshalb in einer Basisklasse implementiert, von der alle anderen Brückenknoten abgeleitet werden. Des Weiteren erhalten die voll polymorphen Brückenknoten eine gemeinsame Basisklasse, in der die Erzeugung und Zerstörung der Anschlüsse zentral implementiert wird.

Die meisten Interaktionsformen existieren einmal als statische und einmal als objektgebundene Variante (z. B. statischer Methodenaufruf und objektgebundener Methodenaufruf). Zusätzlich existieren zwei Paare von Interaktionsformen (Feld lesen, Feld schreiben und Eigenschaft lesen, Eigenschaft schreiben). Dadurch entstehen Interaktionsfamilien. Jede Interaktionsfamilie besitzt eine gemeinsame Basisklasse. So gibt es Basisklassen für zwei methodenbezogene, vier feldbezogene, vier eigenschaftsbezogene und zwei ereignisbezogene Brückenknoten. Die gesamte Vererbungshierarchie ist in *Abbildung 4.12* zu sehen.

Die Beschreibung der Implementierung beginnt mit der Aufteilung der Klassen in zwei Assemblies[2]. Zwei Basisklassen werden in ein eigenes Assembly ausgelagert, um ihre Wiederverwendung zu ermöglichen. Die konkreten Brückenknoten und die Basisklassen der Interaktionsfamilien werden in einem zweiten Assembly implementiert, welches die Knotenbibliothek der Brückenknoten darstellt. Anschließend werden die einzelnen Klassen[3] detailliert beschrieben.

Die Brückenknoten benötigen eine Benutzerschnittstelle für die Konfiguration, in Form von Adressauswahlsteuerelementen[4], mit der eine Interaktion ausgewählt werden kann. Für jede Interaktionsfamilie wird ein eigenes Adressauswahlsteuerelement entworfen.

Auf die Beschreibung der Implementierung folgt das dritte Kapitel des praktischen Anteils: Das Kapitel „Bewertung" (S. 112). Darin werden alle Entwurfsziele einzeln überprüft. Der Implementierungsaufwand ist in jeder Entwurfsentscheidung erfolgreich berücksichtigt worden. Die Struktur der Vererbungshierarchie für die Klassen der Brückenknoten hat wesentlich zu einem niedrigen Implementierungsaufwand beigetragen. Bei der Auswahl der Bindungstechnik wurde eine geringere Performance für eine einfache Implementierung in Kauf ge-

[1] *Unterabschnitt 4.3.7 Klassenhierarchie der Brückenknoten*
[2] *Abschnitt 5.1 Modularisierung*
[3] *Abschnitt 5.2 Klassen für Brückenknoten*
[4] *Abschnitt 5.3 Steuerelemente für die Adressauswahl*

nommen. Die Performance ist akzeptabel (etwa dreifacher Overhead im Vergleich zu einem manuell implementierten Knoten). Sie ließe sich durch eine feste Bindung weiter steigern. Die Gestaltung der Benutzeroberfläche ist selbsterklärend, die Benutzbarkeit ist jedoch verbesserungsfähig. Dies wurde bewusst in Kauf genommen, um den Rahmen der Arbeit nicht zu sprengen.

Den Abschluss der Bewertung bildet eine Aufzählung von Grenzen[1], die der Entwurf und die Implementierung der Brückenknoten besitzen. Dazu zählen fehlende Unterstützung für Indexer (indizierte .NET-Eigenschaften), fehlende Unterstützung für die Auswahl von generischen Klassen und Methoden, keine Möglichkeit Datenflussgraphen oder Teilgraphen als Delegaten zu kapseln, keine Möglichkeit Klassen zu definieren und dabei Schnittstellen zu implementieren oder abstrakte Klassen zu spezialisieren und eine fehlende Unterstützung für eine Ausnahmebehandlung.

Ausblick

Für eine Weiterentwicklung der Brückenknoten können, sowohl aus der Bewertung der einzelnen Entwurfsziele als auch aus den aufgedeckten Grenzen des Entwurfs und der Implementierung, Ansatzpunkte für Verbesserungen und Erweiterungen abgeleitet werden.

Eine erste Verbesserung ist die Implementierung der festen Bindung für Interaktionen. Dazu bestehen die Möglichkeiten entweder einen Generator für IL-Code zu entwerfen oder die Kompatibilität mit dem .NET-Framework 2.0 aufzugeben und die Lambda-Ausdrücke aus dem .NET-Framework 3.5 zu nutzen. Beide Varianten führen zu einem sehr ähnlichen Ergebnis und würden den Overhead der Brückenknoten stark reduzieren.

Die fehlende Unterstützung für Indexer ist ein Schwachpunkt, der sich durch eine Anpassung der Knoten (`AbstractProperty`, `NGet`, `NSet`, `NStaticGet` und `NStaticSet`) beheben lässt. Dazu müssen diese Brückenknoten jedoch zu voll polymorphen Knoten werden, da das .NET-Framework eine variable Anzahl von Index-Parametern für Indexer unterstützt. Evtl. ist eine Mitnutzung der Basisklasse `AbstractMethodBase` möglich, in welcher die Methoden `InitPorts()` und `Work()` bereits eine vollständige Polymorphie unterstützen. Diese Verbesserung sollte als relativ dringend angesehen werden, da nicht nur Arrays sondern auch eine Reihe von wichtigen Container-Klassen in der .NET-Klassenbibliothek Indexer verwenden.

Eine Unterstützung für die Auswahl von generischen Klassen und Methoden ist ebenso wichtig wie die Indexer, da die aktuelle Weiterentwicklung der .NET-Klassenbibliothek zunehmend Gebrauch von generischen Klassen macht. Wichtig sind auch hier Container-Klassen wie `List<T>` oder `Dictionary<TKey, TValue>`.

[1] *Abschnitt 6.5 Grenzen*

Eine Ausgabe von Ausnahmen über einen zusätzlichen Ausgang, sodass eine Ausnahmebehandlung in den Datenflussgraphen möglich wird, sollte ergänzt werden. Sie wird bei den Brückenknoten `NMethod`, `NStaticMethod`, `NConstructor`, `NGet`, `NSet`, `NStaticGet` und `NStaticSet` benötigt. Die Interaktionen, welche sich auf Felder beziehen, und die Überwachung von Ereignissen benötigen keine Ausnahmenbehandlung.

Um die relativ schlechte Benutzbarkeit zu verbessern, sollte eine erweiterte Unterstützung für die Brückenknoten in die DynamicNodes-Entwicklungsumgebung integriert werden. Dazu könnte z. B. ein Browser für das Durchsuchen von .NET-Klassenbibliotheken oder einer Unterstützung durch erweiterte Kontextmenüs an den Ein- und Ausgängen der Knoten gehören.

Literaturverzeichnis

[AbGu04] Abrahams, David ; Gurtovoy, Aleksey: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison-Wesley Professional, 2004. – ISBN 0–3212–2725–5

[Adam70] Adams, D. A.: *A model for parallel computations.* 1970

[Barc90] Barclay, Kenneth A.: *ANSI C.* New York : Prentice Hall, 1990. – 521 S. – ISBN 0–13–037326–5

[BiWa88] Bird, Richard ; Wadler, Philip: *Introduction to Functional Programming.* Prentice Hall, 1988 (Prentice Hall International Series in Computing Science). – 270 S. – ISBN 0–13–484189–1

[Booc91] Booch, Grady: *Object Oriented Design with Applications.* Redwood City, California : Benjamin/Cummings, 1991 (The Benjamin/Cummings series in Ada and Software Engineering). – 580 S.

[Booc94] Booch, Grady: *Object-Oriented Analysis and Design with Applications.* 2nd Edition. Redwood City CA : Benjamin/Cummings, 1994. – 589 S. – ISBN 0805353402

[Boon83] Boon, Kaspar L.: *Basic für Tischcomputer.* München : Pflaum, 1983. – 190 S. – ISBN 3–7905–0375–4

[Denn74] Dennis, J. B.: *First Version of a Data Flow Language.* Springer Verlag, 1974

[Espo] Esposito, Dino: *Die Microsoft Windows Workflow Foundation - Eine Einführung für Entwickler.* `http://msdn.microsoft.com/de-de/library/cc431274`. – Abruf 22.03.2010

[GaHeJoVl94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994. – 416 S. – ISBN 0–2016–3361–2

[Gilo93] Giloi, Wolfgang K.: *Rechnerarchitekturen.* 2. Auflage. Berlin : Springer, 1993. – 482 S. – ISBN 3–540–56355–5

[HaKoHo95] Hanly, Jeri R. ; Koffman, Elliot B. ; Horvath, Joan C.: *C program design for engineers.* Addison-Wesley, 1995. – 725 S. – ISBN 0–201–59064–6

[HuAi04] Hubwieser, Peter ; Aiglstorfer, Gerd: *Fundamente der Informatik*. München : Oldenbourg Wissensch.Vlg, 2004. – 276 S. – ISBN 3–4862–7572–0

[Inte10] Interl Corporation: *Intel Parallel Studio Home*. http://software.intel.com/en-us/intel-parallel-studio-home/. Version: April 2010. – Abruf: 30.04.2010

[Kier07] Kiertscher, Tobias: *DynamicNodes - Entwurf und Implementierung eines Flussgraphen-basierten visuellen Programmiersystems zur parallelen Ausführung auf Mehrprozessorsystemen*, Fachhochschule Brandenburg, Diplomarbeit, August 2007. http://opus.kobv.de/fhbrb/volltexte/2008/42/

[Kier08] Kiertscher, Tobias: *DynamicNodes - Entwurf und Implementierung eines Flussgraphen-basierten visuellen Programmiersystems zur parallelen Ausführung auf Mehrprozessorsystemen*. Saarbrücken : Verlag Dr. Müller, 2008. – 172 S. – ISBN 978–3–639–03362–5. – - Diplomarbeit, Fachhochschule Brandenburg

[Kier09a] Kiertscher, Tobias: *DynamicNodes Projekt-Website - Startseite*. http://dynamicnodes.mastersign.de/. Version: Dezember 2009

[Kier09b] Kiertscher, Tobias: *Merkmale von Datenflusssystemen*. http://informatik.fh-brandenburg.de/~kiertsch/stuff/Studienarbeit_Kiertscher.pdf. Version: April 2009. – Abruf 22.03.2010

[Kosi73] Kosinski, P. R.: *A data flow programming language for operating systems*. 1973

[Lloy87] Lloyd, John W.: *Foundations of logic programming*. 2. Auflage. Springer Verlag, 1987. – 212 S. – ISBN 0387181997

[Math10] MathWorks, The: *The MathWorks Deutschland - Simulink - Simulation und Model-Based-Design*. http://www.mathworks.de/products/simulink/. Version: März 2010. – Abruf 22.03.2010

[Merr08] Merritt, Rick: *CPU designers debate multi-core future*. EE Times. http://www.eetimes.com/showArticle.jhtml?articleID=206105179. Version: Juni 2008. – Abruf 23.03.2010

[Micr09] Microsoft Corporation: *VPL Introduction*. http://msdn.microsoft.com/en-us/library/bb483088. Version: 2009. – Abruf 22.03.2010

[Micr10a] Microsoft Corporation: *Microsoft .NET Framework*. http://www.microsoft.com/NET/. Version: 2010. – Abruf 07.04.2010

[Micr10b] Microsoft Corporation: *Referenz zur .NET-Framework-Klassenbibliothek*. http://msdn.microsoft.com/de-de/library/ms229335.aspx. Version: 2010. – Abruf 06.04.2010

[Micr10c] Microsoft Corporation: *Task Parallel Library*. http://msdn.microsoft.com/en-us/library/dd460717.aspx. Version: April 2010. – Abruf: 30.04.2010

[Mito10] Mitov Software: *OpenWire*. http://www.mitov.com/html/openwire.html. Version: März 2010. – Abruf 22.03.2010

[Mitt97] Mittendorfer, Josef: *SmallTalk*. Addison Wesley, 1997 (2. Auflage). – 500 S.

[Nati10] National Instruments: *NI LabVIEW - The Software That Powers Virtual Instrumentations*. http://www.ni.com/labview/. Version: März 2010. – Abruf 22.03.2010

[Neum93] Neumann, John von: First Draft of a Report on the EDVAC. Version: Oktober 1993. http://dx.doi.org/10.1109/85.238389. Piscataway, NJ, USA : IEEE Educational Activities Department, Oktober 1993. – Forschungsbericht. – 27–75 S.. – ISSN 1058–6180

[Nove10] Novell: *mono – cross platform, open source .NET development framework*. http://www.mono-project.com. Version: 2010. – Abruf 07.04.2010

[Orac10] Oracle: *Developers Resources for Java Technology*. http://java.sun.com. Version: 2010. – Abruf 07.04.2010

[RaRü07] Rauber, Thomas ; Rünger, Gudula: *Multicore:: Parallele Programmierung (Informatik im Fokus)*. deutsche Ausgabe. Springer, 2007. – 164 S. – ISBN 3–5407–3113–X

[Rich06] Richter, Jeffrey: *CLR via C#*. 2nd Edition. Redmond : Microsoft Press, 2006. – 693 S. – ISBN 0–7356–2163–2

[RoHa03] Roy, Peter Van ; Haridi, Seif: *Concepts, Techniques, and Models of Computer Programming*. http://codepoetics.com/wiki/. Version: Juni 2003

[Rumb77] Rumbaugh, J. E.: *A data flow multiprocessor*. 1977

[Sche05] Schellong, Helmuth O. B. ; Schellong, Helmut O. B. (Hrsg.): *Moderne C-Programmierung*. Berlin : Springer, 2005 (Xpert.press). http://dx.doi.org/http://dx.doi.org/10.1007/3-540-28545-8. http://dx.doi.org/http://dx.doi.org/10.1007/3-540-28545-8. – ISBN 3–540–23785–2. – eBook

[ScHuBr06] Schäpers, Arne ; Huttary, Rudolf ; Bremes, Dieter: *C# Kompendium – Window- und Web-Programmierung mit Visual Studio .NET*. Markt und Technik, 2006. – 1159 S.

[Sh+92] Sharp, John A. u. a. ; Sharp, John A. (Hrsg.): *Data flow computing - theory and practice*. Norwood : Ablex Publishing, 1992. – ISBN 0–8939–1654–4

[SyGrCi07] Syme, Don ; Granicz, Adam ; Cisternino, Antonio: *Expert F#*. Apress, 2007 (Expert's Voice in .Net). – 609 S. – ISBN 1–5905–9850–4

[Ters10] Tersus Software Ltd.: *Tersus - As easy as drawing*. http://www.tersus.com/. Version: März 2010. – Abruf 22.03.2010

[Torp08] Torp, Johan: *The parallelism shift and C++'s memory model*. http://www.johantorp.com/parallelism08_cpp_mm.pdf. Version: September 2008. – Abruf 23.03.2010

[TrBrHo82] Treleaven, P. C. ; Brownbridge, D. R. ; Hopkins, R. P.: Data-Driven and Demand-Driven Computer Architecture. In: *ACM Computer Surveys*, 1982

[Unge97] Ungerer, Theo: *Parallelrechner und parallele Programmierung*. Heidelberg : Spektrum Akademischer Verlag, 1997 (Hochschultaschenbuch). – 375 S. – ISBN 3–8274–0231–X. – FHB: TVL-71

[Volk96] Volkmann, L.: *Fundamente der Graphentheorie*. Springer Verlag, 1996

[vvvv10] vvvv group: *vvvv - a multipurpose toolkit*. http://vvvv.org/. Version: März 2010. – Abruf 22.03.2010

[Wa+95] Waldschmidt, Klaus u. a. ; Waldschmidt, Klaus (Hrsg.): *Parallelrechner: Architekturen - Systeme - Werkzeuge*. Stuttgard : B. G. Teubner, 1995 (Leitfäden der Informatik). – 607 S. – ISBN 3–519–02135–8

[WaPa09] Wampler, Dean ; Payne, Alex: *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media, 2009 (Animal Guide). – 448 S. – ISBN 0–5961–5595–6

[Wiki10a] Wikipedia: *C++-Metaprogrammierung — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=C%2B%2B-Metaprogrammierung&oldid=70929902. Version: 2010. – Abruf 23.03.2010

[Wiki10b] Wikipedia: *.NET — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=.NET&oldid=72451881. Version: März 2010. – Abruf: 31.03.2010

Abkürzungsverzeichnis

API Application Programming Interface

CLR Common Language Runtime

COM Component Object Model

CTS Common Type System

DCOM Distributed Component Object Model

GC Garbage Collector

GUI Graphical User Interface

IL Intermediate Language

PC Personal Computer

PDF Portable Document File

UML Unified Modeling Language

URL Uniform Ressource Locator

USB Universal Serial Bus

XML eXtensible Markup Language

A Anhang

A.1 Umfang des Microsoft .NET-Frameworks

Um die Bedeutung einer Einbindung der Klassenbibliotheken des Microsoft .NET-Frameworks zu unterstreichen, soll hier kurz untersucht werden, wie viele Klassen und Mitglieder die Klassenbibliotheken der einzelnen .NET-Framework-Versionen enthalten. Untersucht werden hierzu die Versionen 1.1 bis 4.0 Release Candidate (RC) (vgl. [Wiki10b]). Die Versionen 2.0 und 3.5 werden jeweils inklusive des aktuellen Service Packs 1 untersucht.

A.1.1 Analyse

In dieser kurzen Analyse soll lediglich die Anzahl der Typen in dem .NET-Framework und die Anzahl deren Mitglieder in den verschiedenen Versionen untersucht werden. Die Aussagekraft der reinen Anzahl ist sicherlich stark begrenzt, liefert aber dennoch ein wichtiges Indiz für die Mächtigkeit des Frameworks.

Als Untersuchungshilfsmittel dient die *Microsoft PowerShell* in der Version 2 und ein kleines C#-Programm. Die PowerShell ist eine objektorientierte Befehlszeilenumgebung, die selber auf dem .NET Framework Version 2.0 basiert und sehr gut für Automatisierungsaufgaben geeignet ist. In diesem Testaufbau wird die PowerShell benutzt, um das C#-Programm mehrfach für die verschiedenen Framework-Versionen zu übersetzen. Das Analyse-Programm „FrameworkAnalyzer" wurde so verfasst, dass die C#-Compiler aller Versionen (1.1 bis 4.0) damit umgehen können.

Als Eingabe für den „FrameworkAnalyzer" dienen Listen in Form von Textdateien. Jede Liste enthält alle Assemblies die eine Framework-Version mitbringt. Der FrameworkAnalyzer lädt die in der Liste enthaltenen Assemblies und benutzt die Reflection-API von .NET um die Typen und Typenmitglieder zu zählen.

Zunächst folgt in *Listing A.1* der Quelltext des Programms FrameworkAnalyzer in Form der Datei `Program.cs`.

Listing A.1: FrameworkAnalyzer: Program.cs

```
1 using System;
```

```
using System.IO;
using System.Reflection;

namespace FrameworkAnalyzer
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            string listContent = File.OpenText(args[0]).ReadToEnd();
            string[] list = listContent.Split('\n');

            Console.WriteLine("Scannen␣der␣Assemblies");
            Console.WriteLine("----------------------");

            int assembliesCnt = list.Length;
            int typeCnt = 0;
            int structCnt = 0;
            int interfaceCnt = 0;
            int classCnt = 0;
            int delegateCnt = 0;
            int enumCnt = 0;

            int constructorCnt = 0;
            int methodCnt = 0;
            int propertyCnt = 0;
            int fieldCnt = 0;
            int eventCnt = 0;

            foreach (string listItem in list)
            {
                string assembly = listItem.Trim();
                Assembly a = Assembly.LoadWithPartialName(assembly);
                Type[] types = a.GetTypes();
                typeCnt += types.Length;
                foreach (Type t in types)
                {
                    if (t.IsClass)
                    {
                        if (typeof(Delegate).IsAssignableFrom(t))
                            delegateCnt++;
                        else classCnt++;
                    }
```

```
44                      else if (t.IsInterface) interfaceCnt++;
45                      else if (t.IsEnum) enumCnt++;
46                      else if (!t.IsEnum && !t.IsInterface && !t.IsClass)
                            structCnt++;
47
48                      constructorCnt += t.GetConstructors().Length;
49                      methodCnt += t.GetMethods().Length;
50                      propertyCnt += t.GetProperties().Length;
51                      fieldCnt += t.GetFields().Length;
52                      eventCnt += t.GetEvents().Length;
53                  }
54              }
55              int memberCnt = constructorCnt + methodCnt + propertyCnt +
                    fieldCnt + eventCnt;
56
57              Console.WriteLine();
58              Console.WriteLine("Ergebnisse");
59              Console.WriteLine("----------------------");
60              Console.WriteLine("Assemblies:␣␣␣␣␣{0}", assembliesCnt);
61              Console.WriteLine("Typen:␣␣␣␣␣␣␣␣␣␣{0}", typeCnt);
62              Console.WriteLine("Klassen:␣␣␣␣␣␣␣␣{0}", classCnt);
63              Console.WriteLine("Schnittstellen:␣{0}", interfaceCnt);
64              Console.WriteLine("Strukturen:␣␣␣␣␣{0}", structCnt);
65              Console.WriteLine("Aufzählungen:␣␣␣{0}", enumCnt);
66              Console.WriteLine("Delegaten:␣␣␣␣␣␣{0}", delegateCnt);
67              Console.WriteLine("Mitglieder:␣␣␣␣␣{0}", memberCnt);
68              Console.WriteLine("Konstruktoren:␣␣{0}", constructorCnt);
69              Console.WriteLine("Methoden:␣␣␣␣␣␣␣{0}", methodCnt);
70              Console.WriteLine("Eigenschaften:␣␣{0}", propertyCnt);
71              Console.WriteLine("Felder:␣␣␣␣␣␣␣␣␣{0}", fieldCnt);
72              Console.WriteLine("Ereignisse:␣␣␣␣␣{0}", eventCnt);
73          }
74      }
75  }
```

Da von einer Framework-Version zur nächsten lediglich Assemblies hinzukommen, aus Gründen der Abwärtskompatibilität aber keine entfernt werden, reicht es hier aus die jeweils neuen Assemblies für jede .NET-Framework-Version anzugeben. Die Textdateien, welche als Eingabe für den FrameworkAnalyzer dienen, enthalten jedoch immer alle Assemblies der jeweiligen Framework-Version.

Listing A.2: Assemblies in der Version 1.1.4322

```
1  mscorlib
```

```
System
System.Configuration.Install
System.Data
System.Data.OracleClient
System.Design
System.DirectoryServices
System.Drawing
System.Drawing.Design
System.EnterpriseServices
System.Management
System.Messaging
System.Runtime.Remoting
System.Runtime.Serialization.Formatters.Soap
System.Security
System.ServiceProcess
System.Web
System.Web.Mobile
System.Web.RegularExpressions
System.Web.Services
System.Windows.Forms
System.XML
Microsoft.JScript
Microsoft.VisualBasic
Microsoft.VisualBasic.Vsa
Microsoft.VisualC
Microsoft.Vsa
IEHost
ISymWrapper
```

Listing A.3: Neue Assemblies in der Version 2.0.50727

```
System.Configuration
System.Data.SqlXml
System.Deployment
System.DirectoryServices.Protocols
System.Transactions
Microsoft.Build.Engine
Microsoft.Build.Framework
Microsoft.Build.Tasks
Microsoft.Build.Utilities
Microsoft.VisualBasic.Compatibility
Microsoft.VisualBasic.Compatibility.Data
```

Listing A.4: Neue Assemblies in der Version 3.0.4506

```
System.IdentityModel
System.IdentityModel.Selectors
System.IO.Log
System.Printing
System.Runtime.Serialization
System.ServiceModel
System.ServiceModel.Install
System.ServiceModel.WasHosting
System.Speech
System.Workflow.Activities
System.Workflow.ComponentModel
System.Workflow.Runtime
Microsoft.ServiceModel.Channels.Mail
Microsoft.ServiceModel.Channels.Mail.ExchangeWebService
PresentationBuildTasks
PresentationCore
PresentationFramework
PresentationFramework.Aero
PresentationFramework.Classic
PresentationFramework.Luna
PresentationFramework.Royale
WindowsBase
WindowsFormsIntegration
```

Listing A.5: Neue Assemblies in der Version 3.5.30729

```
System.AddIn
System.AddIn.Contract
System.ComponentModel.DataAnnotations
System.Core
System.Data.DataSetExtensions
System.Data.Entity
System.Data.Entity.Design
System.Data.Linq
System.Data.Services
System.Data.Services.Client
System.Data.Services.Design
System.DirectoryServices.AccountManagement
System.Management.Instrumentation
System.Net
System.ServiceModel.Web
System.Web.Abstractions
System.Web.DynamicData
```

```
System.Web.DynamicData.Design
System.Web.Entity
System.Web.Entity.Design
System.Web.Extensions
System.Web.Extensions.Design
System.Web.Routing
System.Windows.Presentation
System.WorkflowServices
System.Xml.Linq
Microsoft.Build.Conversion.v3.5
Microsoft.Build.Utilities.v3.5
```

Listing A.6: Neue Assemblies in der Version 4.0.30128

```
System.Activities
System.Activities.Core.Presentation
System.Activities.DurableInstancing
System.Activities.Presentation
System.ComponentModel.Composition
System.Device
System.Numerics
System.Runtime.Caching
System.Runtime.DurableInstancing
System.ServiceModel.Activation
System.ServiceModel.Activities
System.ServiceModel.Channels
System.ServiceModel.Discovery
System.ServiceModel.DomainServices.EntityFramework
System.ServiceModel.DomainServices.Hosting
System.ServiceModel.DomainServices.Hosting.OData
System.ServiceModel.DomainServices.Server
System.ServiceModel.Routing
System.Windows.Forms.DataVisualization.Design
System.Windows.Forms.DataVisualization
System.Xaml
Microsoft.Build.Conversion.v4.0
Microsoft.Build.Tasks.v4.0
Microsoft.Build.Utilities.v4.0
Microsoft.CSharp
Microsoft.VisualC.STLCLR
XamlBuildTask
```

Um die Analyse durchzuführen, muss die Datei Program.cs mit dem C#-Compiler jeder .NET-Framework-Version – außer 3.0, denn diese Framework-Version verwendet den Compiler in

Version 2.0 – einmal übersetzt werden. Anschließend wird für jede Version die passende exe-Datei mit der entsprechenden Assembly-Liste aufgerufen. Diese Aufgabe wird in dem PowerShell-Script `Analyze-Frameworks.ps1` (vgl. *Listing A.7*) automatisiert.

Als Rahmenbedingung ist es natürlich erforderlich, dass alle Framework-Versionen auf dem System installiert sind. Das ist nur unter dem Betriebssystem Microsoft Windows XP möglich, da das Framework in der Version 1.1 keine Nachfolger von Windows XP und spätestens die Version 4.0 keine Vorgänger von Windows XP unterstützt.

Listing A.7: Analyze-Frameworks.ps1

```
$csc1 = "C:\Windows\Microsoft.NET\Framework\v1.1.4322\csc.exe"
$csc2 = "C:\Windows\Microsoft.NET\Framework\v2.0.50727\csc.exe"
$csc3 = "C:\Windows\Microsoft.NET\Framework\v3.5\csc.exe"
$csc4 = "C:\Windows\Microsoft.NET\Framework\v4.0.30128\csc.exe"

& $csc1 /warn:0 /out:FrameworkAnalyzer1.exe Program.cs
& $csc2 /warn:0 /out:FrameworkAnalyzer2.exe Program.cs
& $csc3 /warn:0 /out:FrameworkAnalyzer3.exe Program.cs
& $csc4 /warn:0 /out:FrameworkAnalyzer4.exe Program.cs

.\FrameworkAnalyzer1.exe Version-1.1.4322.txt `
        > Ergebnis-1.1.4322.txt

.\FrameworkAnalyzer2.exe Version-2.0.50727.txt `
        > Ergebnis-2.0.50727.txt

.\FrameworkAnalyzer2.exe Version-3.0.4506.txt `
        > Ergebnis-3.0.4506.txt

.\FrameworkAnalyzer3.exe Version-3.5.30729.txt `
        "C:\Program Files (x86)\Reference Assemblies\Microsoft\
            Framework\v3.5" `
        > Ergebnis-3.5.30729.txt

.\FrameworkAnalyzer4.exe Version-4.0.30128.txt `
        "C:\Program Files (x86)\Reference Assemblies\Microsoft\
            Framework\.NETFramework\v4.0" `
        > Ergebnis-4.0.30128.txt
```

Die Ergebnisse sind in der *Tabelle A.1* zusammengefasst und werden in *Abbildung A.1* und *Abbildung A.2* veranschaulicht.

Tabelle A.1: Statistik über die Entwicklung des .NET-Frameworks

	1.1	**2.0 SP1**	**3.0**	**3.5 SP1**	**4.0 RC**
Erschienen	Apr. 2003	Nov. 2005	Nov. 2006	Nov. 2007	Apr. 2010
Assemblies	29	40	632	90	116
Typen	**8426**	**18200**	**31112**	**35863**	**42601**
Klassen	6002	12083	20924	24903	30840
Schnittstellen	663	1284	1883	2071	2421
Strukturen	522	1433	2984	3161	3093
Aufzählungen	961	2716	4102	4436	4726
Delegaten	278	684	1219	1292	1521
Mitglieder	**233145**	**553279**	**919952**	**1007781**	**1264967**
Methoden	160494	385869	637496	700592	881624
Konstruktoren	6010	11187	18544	21003	26265
Felder	20595	38134	70002	75879	84220
Eigenschaften	35328	88690	141063	154305	196533
Ereignisse	10718	29399	52847	56002	76325

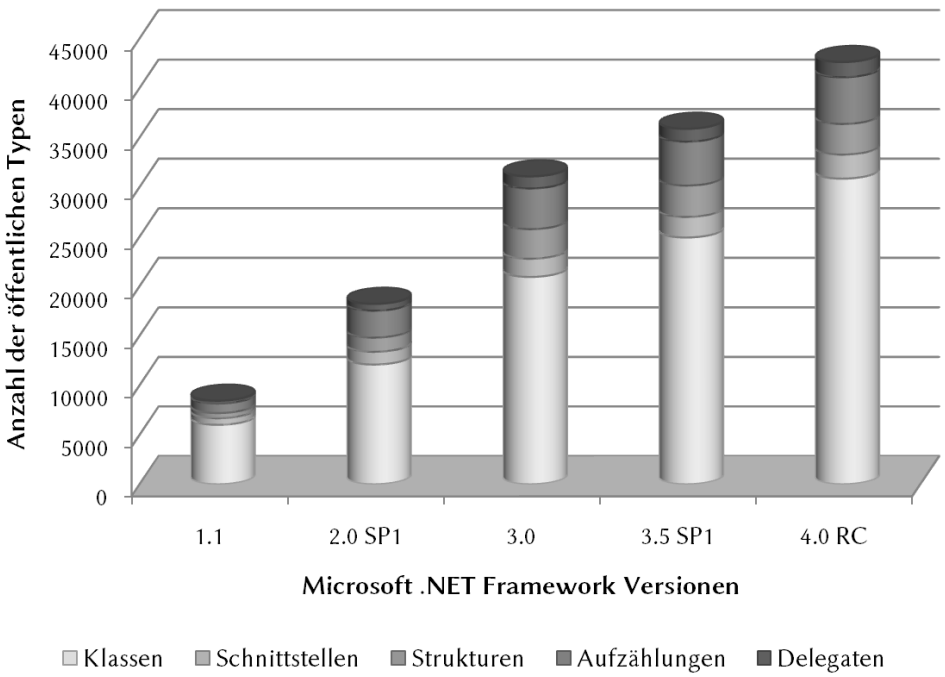


Abbildung A.1: Öffentliche Typen in den .NET-Framework-Versionen

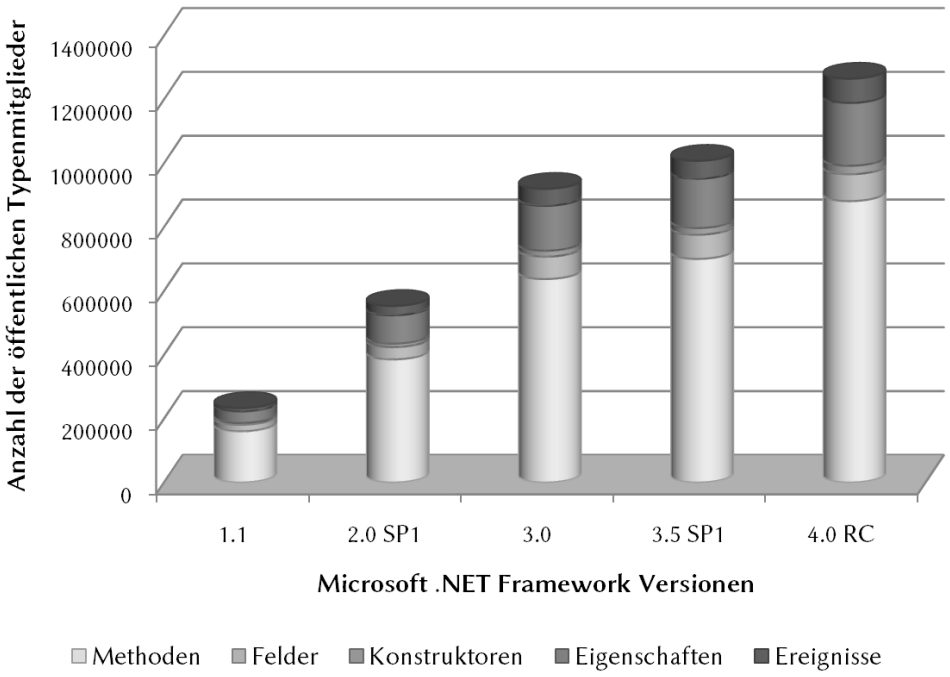


Abbildung A.2: Öffentliche Typenmitglieder in den .NET-Framework-Versionen

A.1.2 Ergebnis

Das Ergebnis der Analyse zeigt, dass das .NET-Framework von Microsoft in den letzten Jahren stark ausgebaut wurde. Die Anzahl der öffentlichen Klassen seit der ersten verbreiteten Version 1.1 (die Version 1.0 lässt sich aufgrund ihrer niedrigen Aktzeptanz seitens der Entwickler vernachlässigen) hat sich von 6002 auf 30840 mehr als verfünffacht. Eine Brücke zwischen einem Datenflusssystem und dieser umfangreichen Klassenbibliothek scheint vielversprechend.

A.2 Benchmark für Bindungstechniken

In dem Microsoft .NET Framework gibt es unterschiedliche Techniken, um eine Methode, die nur über ihre Metadaten bekannt ist, aufzurufen (vgl. *Unterabschnitt 2.3.3 Für diese Arbeit wichtige .NET-Technologien*). In diesem Abschnitt soll die Performance von zwei dieser Techniken gemessen und verglichen werden.

Die erste Technik ist der Aufruf einer Methode über die Reflection-API, die zweite Technik ist der Aufruf über einen zur Laufzeit kompilierten Lambda-Ausdruck.

A.2.1 Analyse

Die Messung der Techniken soll in einem C#-Programm erfolgen (vgl. *Listing A.8*). Jede Messung hat eine Initialisierungsphase (`InitTest<Nr>()`), in der die Bindung des Methodenaufrufs aufgebaut wird, und eine Ausführungsphase (`DoTest<Nr>()`), in der eine Methode mit der jeweiligen Aufruftechnik 10.000.000 mal (`CYCLES`) aufgerufen wird.

Die Messungen mit der Nummer 0A und 0B dienen als Referenz. Die Messung 0A führt keinen Methodenaufruf aus, sondern direkt die gleiche Multiplikationsoperation wie die Testmethode. Die Messung 0B verwendet eine direkte Bindung zur Kompilierzeit. Die direkte Bindung ist ein gewöhnlicher Methodenaufruf für den Fall, dass eine Methode zur Entwicklungszeit bekannt ist.

Die Messung mit der Nummer 1 ermittelt in der ersten Phase das `MethodInfo`-Objekt für die aufzurufende Methode und speichert diese zwischen. In der Ausführungsphase wird die Methode mit Hilfe von `MethodInfo.Invoke()` aufgerufen.

Die Messung mit der Nummer 2 ermittelt in der ersten Phase ebenfalls das MethodInfo-Objekt für die aufzurufende Methode, setzt aber anschließend einen Lambda-Ausdruck zusammen (`Expression.Call()`, `Expression.Parameter()`, `Expression.Lambda()`) und kompiliert diesen mit einem Aufruf von `Expression<T>.Compile()`. Das Ergebnis der Kompilierung ist ein Delegat (typisierter Funktionszeiger), welcher zwischengespeichert wird. In der Ausführungsphase wird der zwischengespeicherte Delegat direkt aufgerufen.

Die aufgerufene Methode führt lediglich eine Multiplikation von zwei 32-Bit Ganzzahlen durch, damit in der Messung der Overhead für den Aufruf dominant ist.

Das Programm wurde im *Debug*-Modus übersetzt, um zu verhindern, dass der C#-Compiler den Aufruf der Testmethode im Referenztest mit einer „Inlining" genannten Technik entfernt. Wird dem Compiler erlaubt diese Optimierungstechnik anzuwenden, entfällt der Overhead für den Methodenaufruf vollständig und der Test 0 ist nicht mehr als Referenztest zu gebrauchen.

Die Messung der Zeitspannen erfolgt mit der .NET-Klasse `System.Diagnostics.Stopwatch`.

Listing A.8: Programm zur Messung der Performance von Methodenaufruftechniken

```
1  using System;
2  using System.Diagnostics;
3  using System.Reflection;
4  using System.Linq.Expressions;
5
6  namespace de.mastersign.demo
7  {
8      class TestClass
```

```
9     {
10        public int TestMethod(int a, int b)
11        {
12            return a * b;
13        }
14    }
15
16    class Program
17    {
18        private const int CYCLES = 10000000;
19        private static object instance;
20
21        public static void Main(string[] args)
22        {
23            instance = new TestClass();
24
25            var sw = new Stopwatch();
26            Console.WriteLine("Durchläufe:␣{0}", CYCLES);
27            Console.WriteLine("Zum␣Starten␣eine␣Taste␣drücken...");
28            Console.ReadKey();
29            System.Threading.Thread.Sleep(1000);
30
31            Console.WriteLine("\nTest:␣Ohne␣Aufruf");
32            sw.Start();
33            DoTest0A();
34            sw.Stop();
35            Console.WriteLine("Zeit␣=␣{0}", sw.Elapsed);
36
37            Console.WriteLine("\nTest:␣Direkt");
38            sw.Start();
39            DoTest0B();
40            sw.Stop();
41            Console.WriteLine("Zeit␣=␣{0}", sw.Elapsed);
42
43            Console.WriteLine("\nTest:␣Reflection-API");
44            InitTest1();
45            sw.Reset();
46            sw.Start();
47            DoTest1();
48            sw.Stop();
49            Console.WriteLine("Zeit␣=␣{0}", sw.Elapsed);
50
51            Console.WriteLine("\nTest:␣Lambda-Expression");
```

```
52              InitTest2();
53              sw.Reset();
54              sw.Start();
55              DoTest2();
56              sw.Stop();
57              Console.WriteLine("Zeit␣=␣{0}", sw.Elapsed);
58          }
59
60          public static void DoTest0A()
61          {
62              int a = 20, b = 120;
63              int result = 0;
64              for (int i = 0; i < CYCLES; i++)
65              {
66                  result = a * b;
67              }
68              Console.WriteLine("Ergebnis␣=␣{0}", result);
69          }
70
71          public static void DoTest0B()
72          {
73              int a = 20, b = 120;
74              int result = 0;
75              var knownInstance = (TestClass)instance;
76              for (int i = 0; i < CYCLES; i++)
77              {
78                  result = knownInstance.TestMethod(a, b);
79              }
80              Console.WriteLine("Ergebnis␣=␣{0}", result);
81          }
82
83          private static MethodInfo mi1;
84
85          public static void InitTest1()
86          {
87              mi1 = instance.GetType().GetMethod("TestMethod");
88          }
89
90          public static void DoTest1()
91          {
92              int a = 20, b = 120;
93              int result = 0;
94              for (int i = 0; i < CYCLES; i++)
```

```
95              {
96                  result = (int)mi1.Invoke(instance, new object[] { a, b
                        });
97              }
98              Console.WriteLine("Ergebnis␣=␣{0}", result);
99          }
100
101         private static MethodInfo mi2;
102         private static Func<int, int, int> d2;
103
104         public static void InitTest2()
105         {
106             mi2 = instance.GetType().GetMethod("TestMethod");
107
108             var parameters = new[] { Expression.Parameter(typeof(int),
                    "a"), Expression.Parameter(typeof(int), "b") };
109             var call = Expression.Call(Expression.Constant(instance),
                    mi2, parameters);
110             var lambda = Expression.Lambda<Func<int, int, int>>(call,
                    parameters);
111             d2 = lambda.Compile();
112         }
113
114         public static void DoTest2()
115         {
116             int a = 20, b = 120;
117             int result = 0;
118             for (int i = 0; i < CYCLES; i++)
119             {
120                 result = d2(a, b);
121             }
122             Console.WriteLine("Ergebnis␣=␣{0}", result);
123         }
124     }
125 }
```

Die Ausgabe des Programms in *Listing A.8* ist in *Listing A.9* dargestellt.

Listing A.9: Ergebnisse des Programms in Listing A.8

```
1 Durchläufe: 10000000
2 Zum Starten eine Taste drücken...
3
4 Test: Ohne Aufruf
5 Ergebnis = 2400
```

```
Zeit = 00:00:00.0331246

Test: Direkt
Ergebnis = 2400
Zeit = 00:00:00.1019227

Test: Reflection-API
Ergebnis = 2400
Zeit = 00:00:31.1425107

Test: Lambda-Expression
Ergebnis = 2400
Zeit = 00:00:00.1205543
```

A.2.2 Testumgebung

Es werden zwei Aufruftechniken gegenübergestellt. Die Zeit für den Aufbau der Bindung wird nicht in die Messung mit einbezogen. Der Methodenaufruf wird mit jeder Aufruftechnik 10.000.000 mal durchgeführt, um eine vergleichbar große Zeitspanne zu erhalten. Das Programm wurde auf einem PC mit Intel Core i7 860 unter Microsoft Windows 7 und Microsoft .NET 3.5 ausgeführt.

Bei der Ausführung wurde die Prozessoraffinität auf 6 und 7 gesetzt. Jeder Kern des Core i7 860 kann durch Hyperthreading zwei Threads ausführen. Thread 6 und 7 bilden zusammen den Kern 3. Damit wird das Programm ausschließlich auf einem der vier Kerne ausgeführt und Verfälschungen, die durch ein *Hopping* des Prozesses zwischen Prozessorkernen entstehen, werden vermieden.

A.2.3 Ergebnis

Alle folgenden gemessenen Zeitspannen beziehen sich auf eine zehnmillionenfache Ausführung. Die Multiplikation ohne Methodenaufruf hat 33 Millisekunden benötigt. Diese Zeitspanne kann von den weiteren Messungen abgezogen werden, da sie den Zeitbedarf für die Ausführung der Methode darstellen und für den Vergleich nur der Overhead interessant ist, der durch den Aufruf entsteht. Direkte Aufrufe der Testmethode haben $102 - 33 = 69$ Millisekunden benötigt. Die Aufrufe mit Hilfe der Reflection-API, haben $31143 - 33 = 31110$ Millisekunden benötigt. Die Aufrufe mit Hilfe eines vorkompilierten Lambda-Ausdrucks haben $121 - 33 = 88$ Millisekunden benötigt.

Um die Ergebnisse vergleichen zu können, werden sie auf die Referenzmessung aus Test 0B normiert. Dabei ergeben sich die in *Tabelle A.2* dargestellten Werte.

Tabelle A.2: Ergebnisse des Performance-Tests von Methodenaufruftechniken

	Zeitspanne in ms	Overhead in ms	normiert
Direkter Aufruf	102	69	1,0
Reflection-API	31143	31110	451,9
Lambda-Ausdruck	121	88	1,3

Der indirekte Aufruf einer Methode, die nur durch ihre Metadaten (`MethodInfo`-Objekt) bekannt ist, erzeugt mit der Reflection-API einen um zwei Größenordnungen höheren Overhead, als mit einem zur Laufzeit kompilierten Lambda-Ausdruck. Der Overhead, der durch den indirekten Aufruf mittels eines zur Laufzeit kompilierten Lambda-Ausdrucks erzeugt wird, ist lediglich 1,3-fach so groß wie der Overhead eines normalen Methodenaufrufs.

A.3 Diagramme

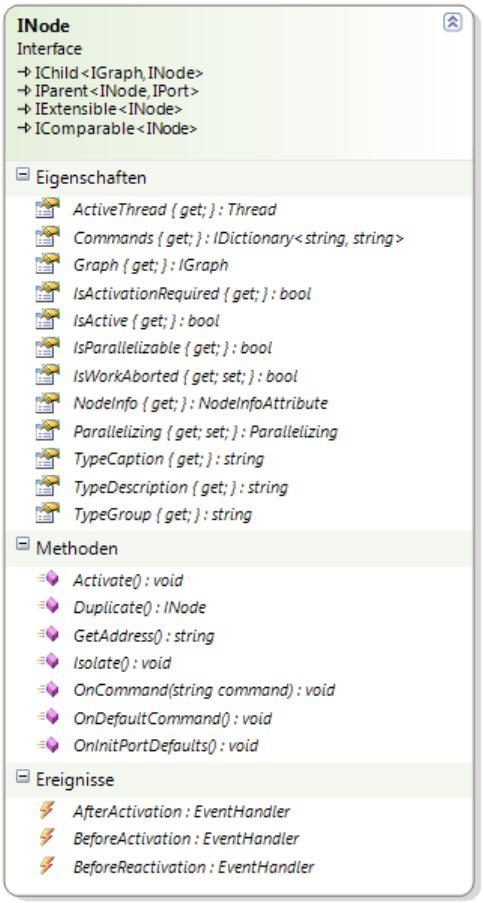


Abbildung A.3: DynamicNode.Core.INode

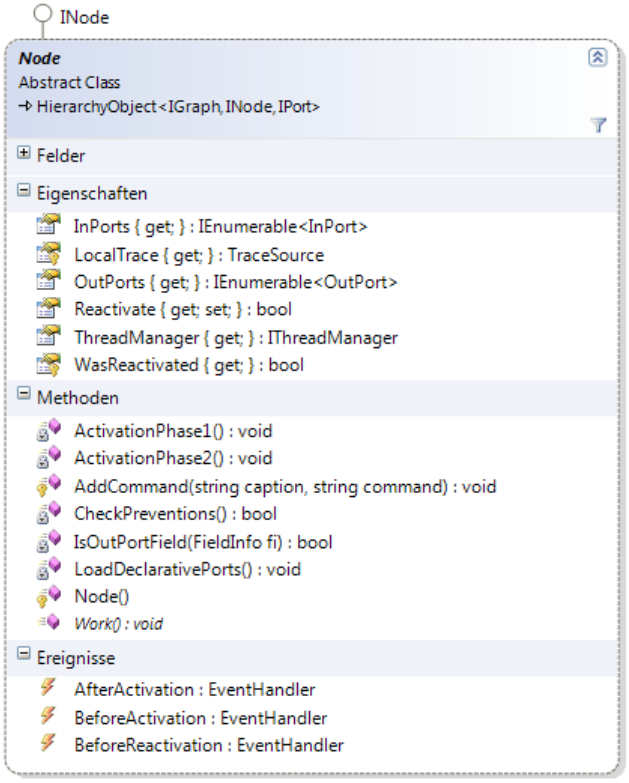


Abbildung A.4: DynamicNode.Core.Node

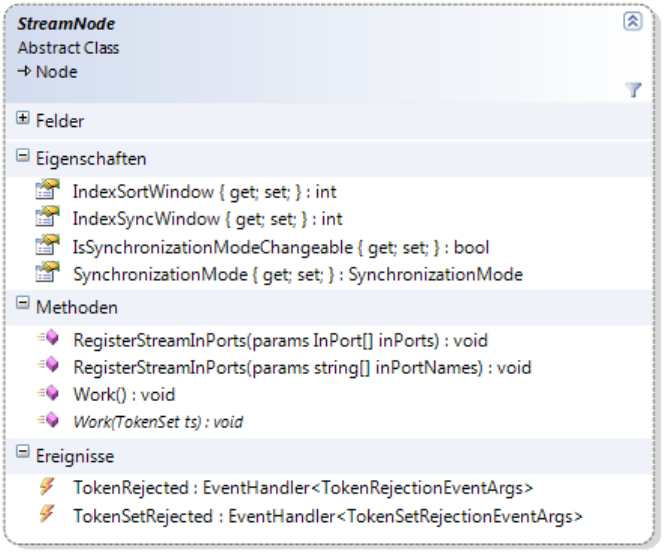


Abbildung A.5: DynamicNode.Core.StreamNode

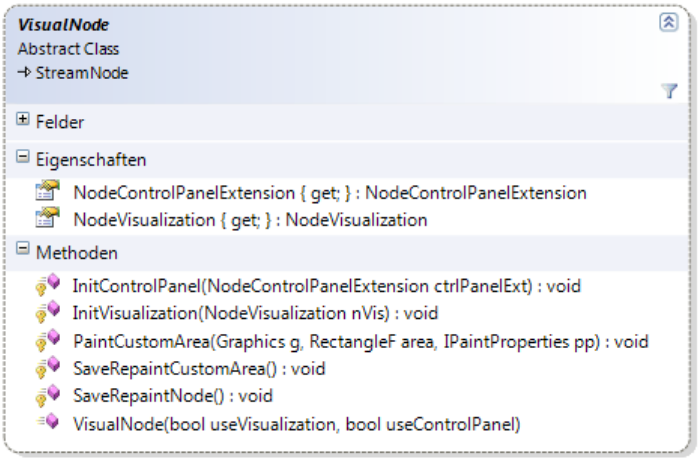


Abbildung A.6: DynamicNode.Ext.Visual.VisualNode

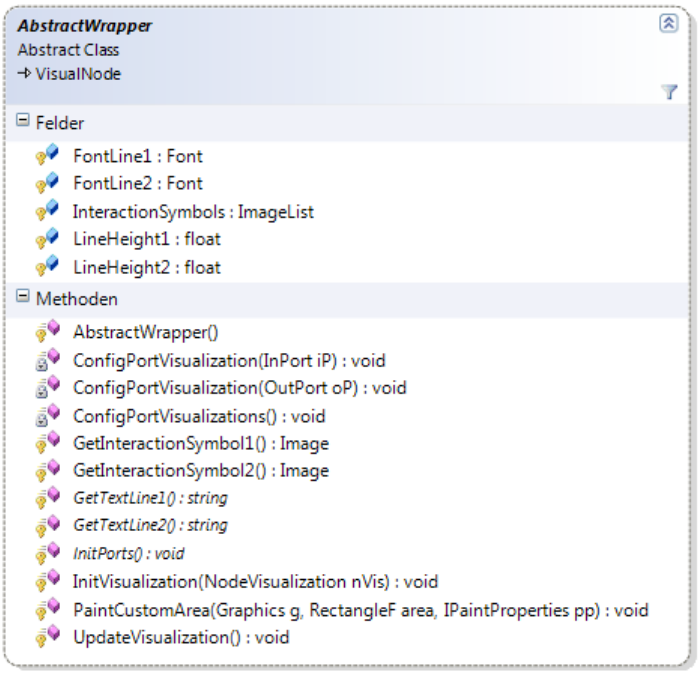


Abbildung A.7: DynamicNode.Ext.Objects.AbstractWrapper

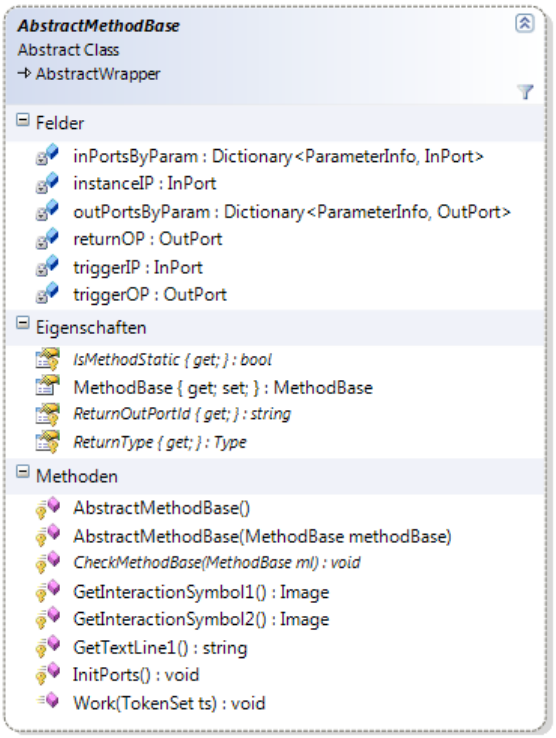


Abbildung A.8: DynamicNode.Ext.Objects.AbstractMethodBase

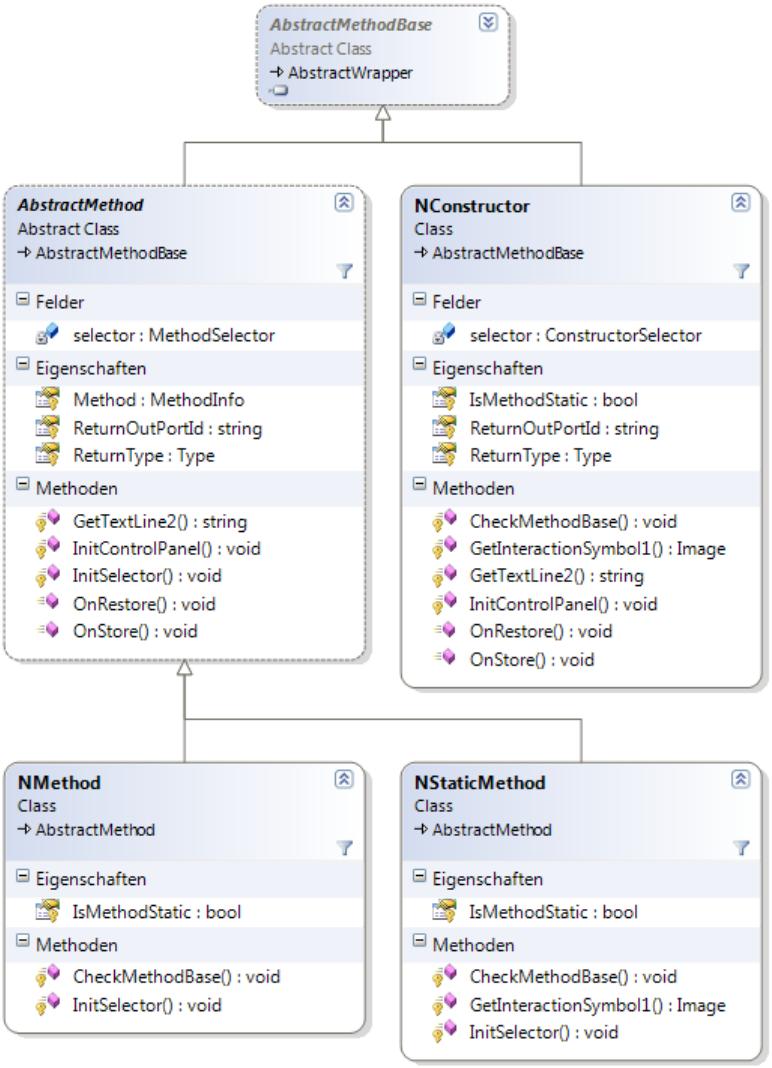


Abbildung A.9: Brückenknoten für Methoden

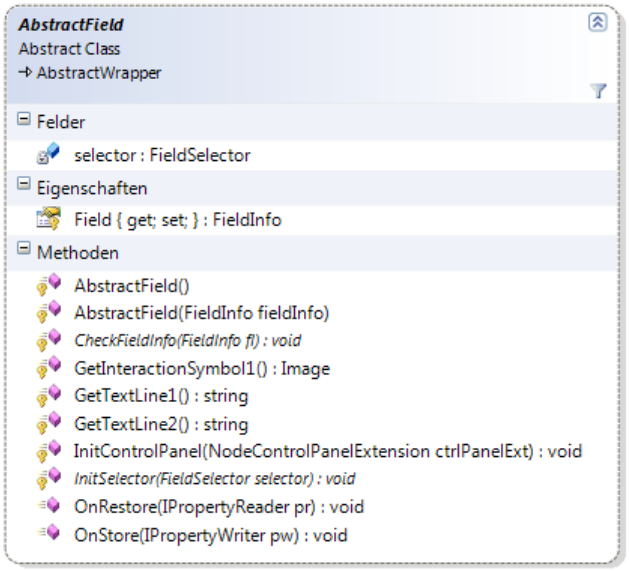


Abbildung A.10: DynamicNode.Lib.Objects.AbstractField

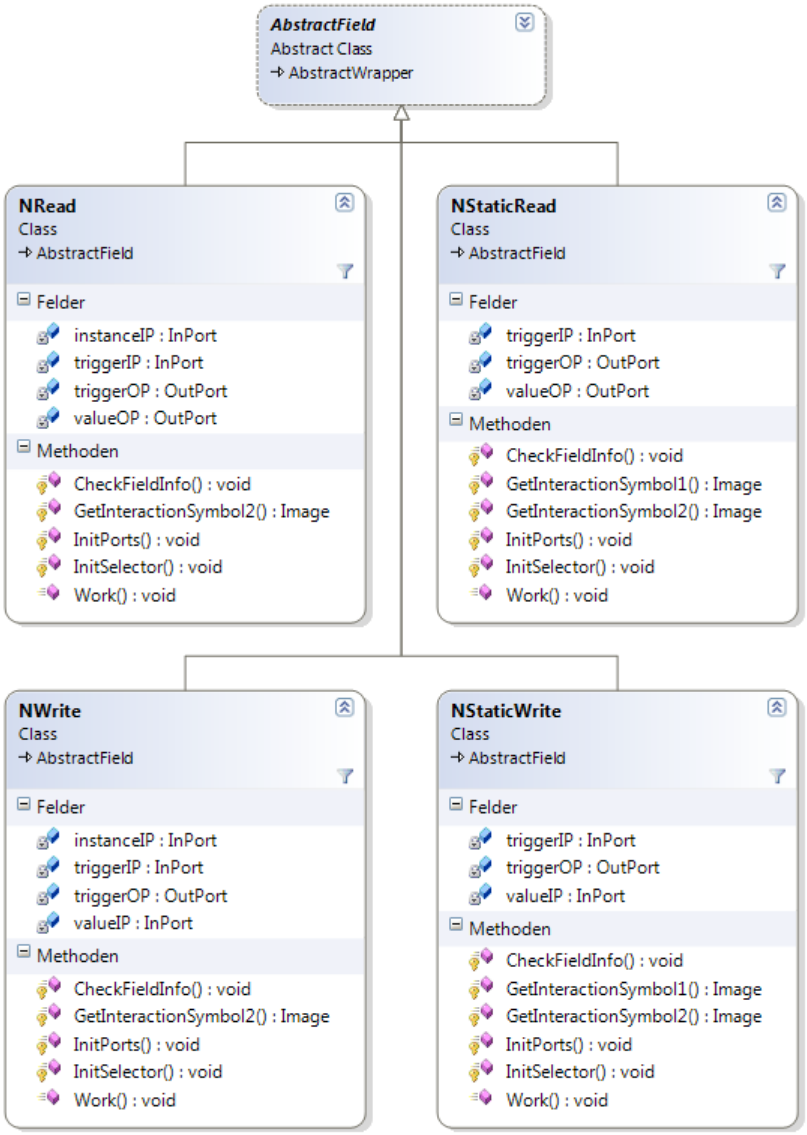


Abbildung A.11: Brückenknoten für Felder

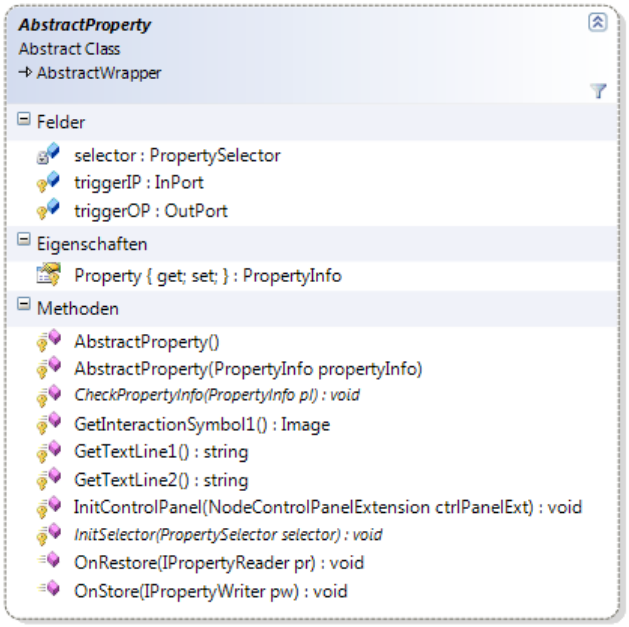


Abbildung A.12: DynamicNode.Lib.Objects.AbstractProperty

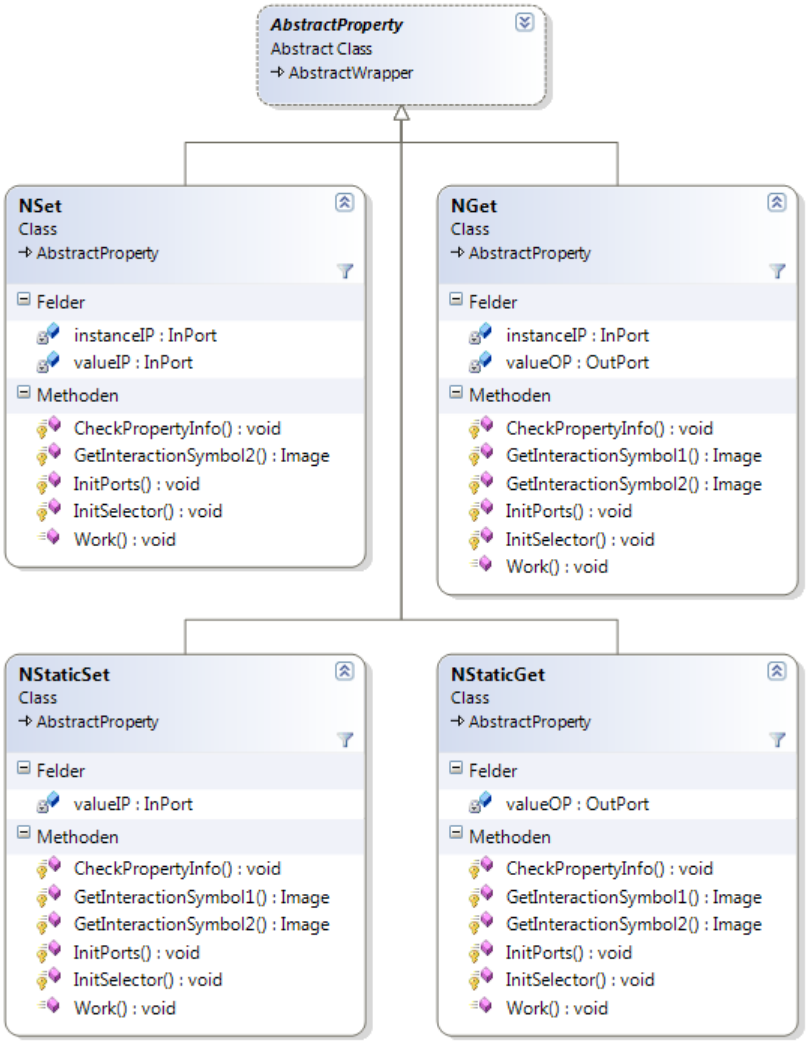


Abbildung A.13: Brückenknoten für Eigenschaften

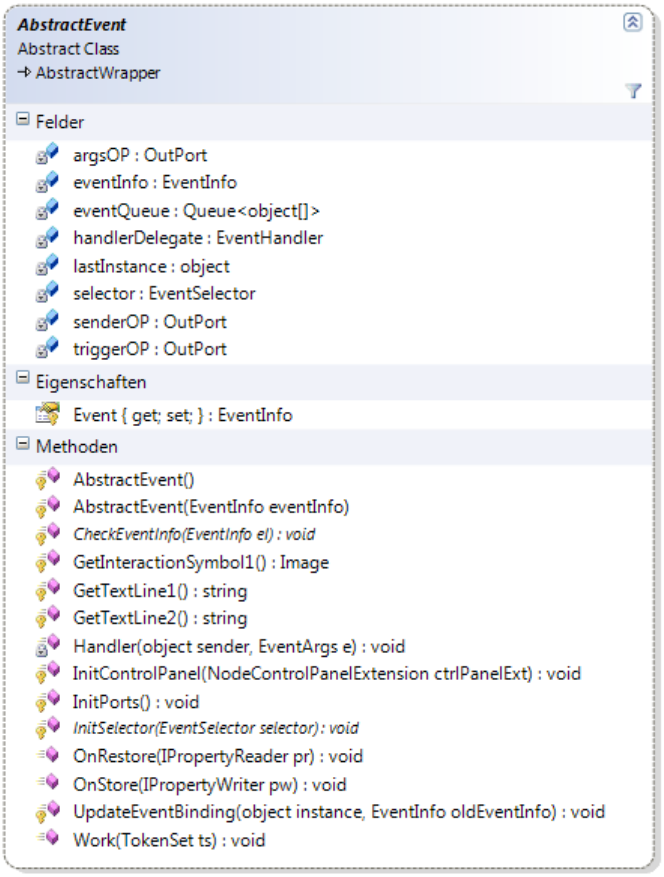


Abbildung A.14: DynamicNode.Lib.Objects.AbstractEvent

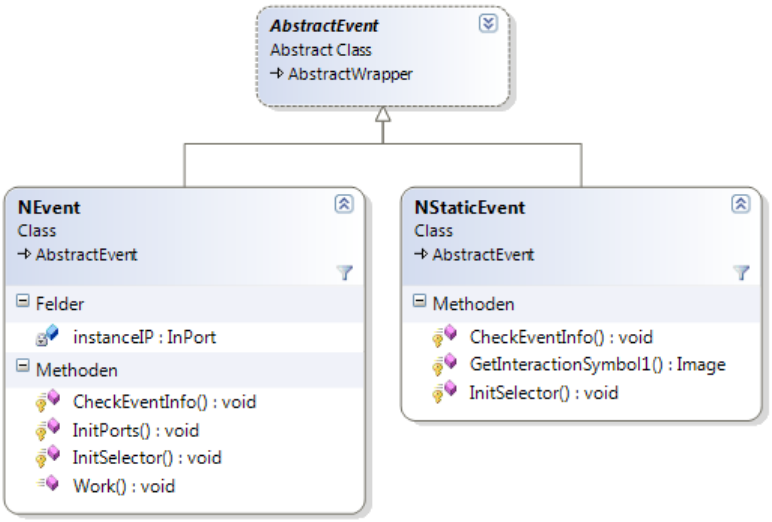


Abbildung A.15: Brückenknoten für Ereignisse

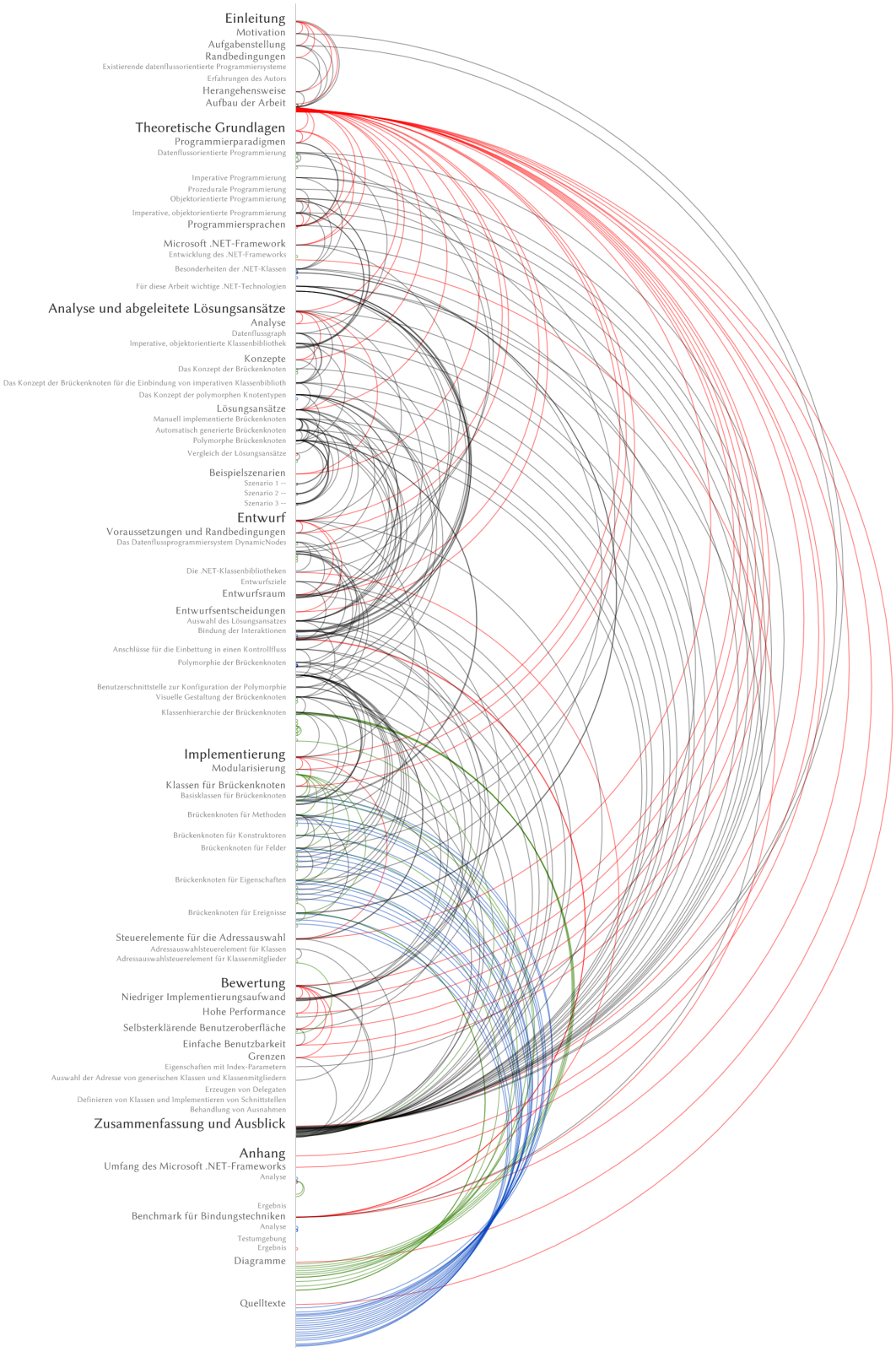


Abbildung A.16: Querverweise innerhalb der Arbeit

A.4 Quelltexte

Listing A.10: DynamicNode.Ext.Objects.AbstractWrapper

```
#region file header

// <copyright>
// Copyright (c) 2009-2010 by Tobias Kiertscher
// Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
// </copyright>
// <license>DNLicense.html</license>
// <revision>$Revision: 1476 $</revision>
// <author>$Author: Kiertscher.Tobias $</author>
// <date>$Date: 2010-04-26 15:56:29 +0200 (Mo, 26. Apr 2010) $</date>

#endregion

using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;
using DynamicNode.Core;
using DynamicNode.Ext.Visual;
using System.Reflection;

namespace DynamicNode.Ext.Objects
{
    /// <summary>
    /// Die Basisklasse für alle CLR-Wrapper-Knoten.
    /// </summary>
    public abstract class AbstractWrapper : VisualNode
    {
        /// <summary>
        /// Ein <see cref="System.Windows.Forms.ImageList"/>-Objekt, welches typische .NET-
            Element-Symbole enthält.
        /// </summary>
        protected static readonly ImageList InteractionSymbols = ClrTools.
            CreateInteractionSymbolList();

        /// <summary>
        /// Die Standardschriftart für die erste Textzeile in der Knotendarstellung.
        /// </summary>
        protected static readonly Font FontLine1 = new Font("Tahoma", 7f);

        /// <summary>
        /// Die Standardschriftart für die zweite Textzeile in der Knotendarstellung.
        /// </summary>
        protected static readonly Font FontLine2 = new Font("Tahoma", 8f);

        /// <summary>
        /// Die Standardschriftgröße für die erste Textzeile in der Knotendarstellung.
        /// </summary>
        protected static readonly float LineHeight1 = 13f;

        /// <summary>
        /// Die Standardschriftgröße für die zweite Textzeile in der Knotendarstellung.
        /// </summary>
        protected static readonly float LineHeight2 = 15f;

        /// <summary>
        /// Ein Bild von dem <see cref="Graphics"/>-Objekte erstellt werden können
        /// um Schriftgrößenmessungen durch zu führen.
        /// </summary>
        protected static readonly Bitmap FontMeasureBmp = new Bitmap(16, 16, PixelFormat.
            Format24bppRgb);

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="AbstractWrapper"/>.
        /// </summary>
        protected AbstractWrapper()
            : base(true, true)
        {
            Init();
        }

        /// <summary>
        /// Initialisiert die aktuelle Instanz.
        /// Wird von den Konstruktoren aufgerufen.
        /// </summary>
        protected virtual void Init()
        {
            AttachExtension(new InPortPersistenceExt());
            AddCommand("Ausführen", "work");
        }

        /// <summary>
```

```
80          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
81          /// </summary>
82          protected abstract void InitPorts();
83
84          /// <summary>
85          /// Wird aufgerufen, wenn der Benutzer den Standardbefehl des Knotens
86          /// aufgerufen hat.
87          /// </summary>
88          /// <remarks>
89          /// Diese Methode muss nicht überschrieben werden.
90          /// </remarks>
91          public override void OnDefaultCommand()
92          {
93              OnCommand("work");
94          }
95
96          /// <summary>
97          /// Wird aufgerufen wenn der Benutzer einen Benutzerbefehl aufgerufen hat.
98          /// </summary>
99          /// <param name="command">Der Befehls-Code des Benutzerbefehls.</param>
100         /// <remarks>
101         /// Diese Methode sollte von einer Klasse überschrieben werden,
102         /// die einen oder mehrere Benutzerbefehle mit <see cref="Node.AddCommand"/>
103         /// registriert hat.
104         /// </remarks>
105         public override void OnCommand(string command)
106         {
107             if (command == "work")
108             {
109                 Activate();
110             }
111         }
112
113         /// <summary>
114         /// Initialisiert die Knoten-Visualisierung.
115         /// </summary>
116         /// <param name="nVis">Die Knoten-Visualisierung.</param>
117         protected override void InitVisualization(NodeVisualization nVis)
118         {
119             base.InitVisualization(nVis);
120             nVis.ShowName = false;
121             ConfigPortVisualizations();
122
123             nVis.ClientHeight = LineHeight1 + LineHeight2 + 4f;
124             nVis.ClientWidth = GetMaxClientWidth();
125         }
126
127         /// <summary>
128         /// Aktualisiert die visuelle Darstellung der Anschlüsse.
129         /// </summary>
130         protected void UpdateVisualization()
131         {
132             var nVis = NodeVisualization;
133             if (nVis == null)
134             {
135                 return;
136             }
137
138             nVis.ClientWidth = GetMaxClientWidth();
139             ConfigPortVisualizations();
140             nVis.ArrangePorts();
141             nVis.RepaintNode();
142         }
143
144         private void ConfigPortVisualizations()
145         {
146             foreach (var iP in InPorts)
147             {
148                 ConfigPortVisualization(iP);
149             }
150             foreach (var oP in OutPorts)
151             {
152                 ConfigPortVisualization(oP);
153             }
154         }
155
156         private static void ConfigPortVisualization(InPort iP)
157         {
158             PortVisualization pv;
159             if (iP.HasExtension(PortVisualization.EXT_ID))
160             {
161                 pv = (PortVisualization)iP.GetExtension(PortVisualization.EXT_ID);
162             }
163             else
164             {
165                 pv = new PortVisualization();
166                 iP.AttachExtension(pv);
167             }
```

```
168             pv.Orientation =
169                 (iP.Name.Equals("trigger") || iP.Name.Equals("triggerIn"))
170                     ? PortOrientation.LEFT : PortOrientation.TOP;
171         }
172 
173         private static void ConfigPortVisualization(OutPort oP)
174         {
175             PortVisualization pv;
176             if (oP.HasExtension(PortVisualization.EXT_ID))
177             {
178                 pv = (PortVisualization)oP.GetExtension(PortVisualization.EXT_ID);
179             }
180             else
181             {
182                 pv = new PortVisualization();
183                 oP.AttachExtension(pv);
184             }
185             pv.Orientation =
186                 (oP.Name.Equals("trigger") || oP.Name.Equals("triggerOut"))
187                     ? PortOrientation.LEFT : PortOrientation.BOTTOM;
188         }
189 
190         /// <summary>
191         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
192         /// welches das gekapselte .NET-Element repräsentieren.
193         /// </summary>
194         /// <returns>Das erste von zwei Symbolen.</returns>
195         protected virtual Image GetInteractionSymbol1()
196         {
197             return InteractionSymbols.Images["Class"];
198         }
199 
200         /// <summary>
201         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
202         /// welches das gekapselte .NET-Element repräsentieren.
203         /// </summary>
204         /// <returns>Das zweite von zwei Symbolen oder <c>null</c>.</returns>
205         protected virtual Image GetInteractionSymbol2() { return null; }
206 
207         /// <summary>
208         /// Gibt die erste von drei möglichen Textzeilen zurück,
209         /// welche das gekapselte .NET-Element beschreiben.
210         /// </summary>
211         /// <returns>Die erste von drei Textzeilen.</returns>
212         protected abstract string GetTextLine1();
213 
214         /// <summary>
215         /// Gibt die zweite von drei möglichen Textzeilen zurück,
216         /// welche das gekapselte .NET-Element beschreiben.
217         /// </summary>
218         /// <returns>Die zweite von drei Textzeilen.</returns>
219         protected abstract string GetTextLine2();
220 
221         /// <summary>
222         /// Wir aufgerufen, um den benutzerdefinierten Bereich des Knotens zu zeichnen.
223         /// </summary>
224         /// <param name="g">Der grafische Kontext.</param>
225         /// <param name="area">Der benutzerdefinierte Bereich.</param>
226         /// <param name="pp">Eine Sammlung von Zeichenressourcen.</param>
227         protected override void PaintCustomArea(Graphics g, RectangleF area,
                 IPaintProperties pp)
228         {
229             g.FillRectangle(pp.NodeBackgroundBrush, area);
230             var img1 = GetInteractionSymbol1();
231             var img2 = GetInteractionSymbol2();
232             float l = area.Left + 2f;
233             float t = area.Top + 2f;
234             g.DrawString(GetTextLine1(), FontLine1, pp.NodeShaddowBrush, l, t);
235             t += LineHeight1;
236             if (img1 != null)
237             {
238                 g.DrawImage(img1, l, t);
239                 l += img1.Width + 1f;
240             }
241             if (img2 != null)
242             {
243                 g.DrawImage(img2, l, t);
244                 l += img2.Width;
245             }
246             t += 1f;
247             g.DrawString(GetTextLine2(), FontLine2, pp.NodeForegroundBrush, l, t);
248             //t += lineHeight2;
249         }
250 
251         /// <summary>
252         /// Gibt die maximale Breite des benutzerdefinierten
253         /// Bereichs der Knotendarstellung zurück.
254         /// Die Breite wird anhand der Darstellung von den zwei Symbolen
```

```
255         /// und den drei Textzeilen ermittelt.
256         /// </summary>
257         /// <returns>Die maximale Breite der benutzerdefinierten Knotendarstellung.</
                returns>
258         protected float GetMaxClientWidth()
259         {
260             var img1 = GetInteractionSymbol1();
261             var img2 = GetInteractionSymbol2();
262             float w1 = 6f;
263             float w2 = 6f;
264             if (img1 != null)
265             {
266                 w2 += img1.Width + 1f;
267             }
268             if (img2 != null)
269             {
270                 w2 += img2.Width + 1f;
271             }
272             using (var g = Graphics.FromImage(FontMeasureBmp))
273             {
274                 string l1 = GetTextLine1();
275                 string l2 = GetTextLine2();
276                 if (!string.IsNullOrEmpty(l1))
277                 {
278                     w1 += g.MeasureString(l1, FontLine1).Width;
279                 }
280                 if (!string.IsNullOrEmpty(l2))
281                 {
282                     w2 += g.MeasureString(l2, FontLine2).Width;
283                 }
284             }
285             return Math.Max(Math.Max(w1, w2), 64f);
286         }
287     }
288 }
```

Listing A.11: DynamicNode.Ext.Objects.AbstractMethodBase

```
1  #region file header
2
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1529 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-21 13:14:11 +0200 (Fr, 21. Mai 2010) $</date>
11
12 #endregion
13
14 using System;
15 using System.Collections.Generic;
16 using System.Drawing;
17 using System.Reflection;
18 using DynamicNode.Core;
19
20 namespace DynamicNode.Ext.Objects
21 {
22     /// <summary>
23     /// Abstrakte Basisklasse für Knoten, deren Aufgabe darin besteht eine Methode
24     /// auf zu rufen.
25     /// Die Methode wird mit einer <see cref="MethodInfo"/> an den Knoten gebunden.
26     /// Die Anschlüsse des Knotens werden automatisch der Methode entsprechende
            eingerichtet.
27     /// </summary>
28     public abstract class AbstractMethodBase : AbstractWrapper
29     {
30         private MethodBase methodBase;
31
32         private readonly Dictionary<ParameterInfo, InPort> inPortsByParam
33             = new Dictionary<ParameterInfo, InPort>();
34
35         private readonly Dictionary<ParameterInfo, OutPort> outPortsByParam
36             = new Dictionary<ParameterInfo, OutPort>();
37
38         private InPort triggerIP;
39         private InPortSingleDataTypeProvider instanceTypeProvider;
40         private InPort instanceIP;
41         private OutPortDataTypeProvider returnTypeProvider;
42         private OutPort returnOP;
43         private OutPort triggerOP;
44
45         /// <summary>
46         /// Initialisiert eine neue Instanz von <see cref="AbstractMethodBase"/>.
47         /// </summary>
```

```
48          protected AbstractMethodBase()
49          {
50          }
51
52          /// <summary>
53          /// Initialisiert eine neue Instanz von <see cref="AbstractMethodBase"/>.
54          /// </summary>
55          /// <param name="methodBase">Eine initiale CLR-Methode.</param>
56          protected AbstractMethodBase(MethodBase methodBase)
57          {
58              CheckMethodBase(methodBase);
59              this.methodBase = methodBase;
60              InitPorts();
61          }
62
63          /// <summary>
64          /// Liest oder schreibt die gekapselte Methode.
65          /// </summary>
66          /// <value>Die gekapselte CLR-Methode.</value>
67          public MethodBase MethodBase
68          {
69              get { return methodBase; }
70              set
71              {
72                  if (methodBase == value)
73                  {
74                      return;
75                  }
76                  if (value != null)
77                  {
78                      try
79                      {
80                          CheckMethodBase(value);
81                      }
82                      catch (ArgumentException e)
83                      {
84                          // TODO String auslagern
85                          TraceWarning(422, "Ungültige␣Methode␣ausgewählt.", e);
86                      }
87                  }
88                  methodBase = value;
89                  InitPorts();
90              }
91          }
92
93          /// <summary>
94          /// Überprüft ob die übergebene Methode geeignet ist.
95          /// Falls die Methode ungeeignet ist wird eine <see cref="ArgumentException"/>
96          /// geworfen.
97          /// </summary>
98          /// <param name="mI">Die zu prüfende Methode.</param>
99          protected abstract void CheckMethodBase(MethodBase mI);
100
101         /// <summary>
102         /// Initialisiert die aktuelle Instanz.
103         /// Wird von den Konstruktoren aufgerufen.
104         /// </summary>
105         protected override void Init()
106         {
107             base.Init();
108             triggerIP = new InPort(this, "triggerIn");
109             instanceTypeProvider = new InPortSingleDataTypeProvider();
110             returnTypeProvider = new OutPortDataTypeProvider();
111             triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
112         }
113
114         /// <summary>
115         /// Gibt die Port-ID für den Ausgang zurück,
116         /// der Rückgabewerte des Methodenaufrufs weiterleiten soll.
117         /// </summary>
118         protected abstract string ReturnOutPortId { get; }
119
120         /// <summary>
121         /// Gibt den Datentyp für den Ausgang zurück,
122         /// der den Rückgabewert des Methodenaufrufs weiterleiten soll.
123         /// </summary>
124         protected abstract Type ReturnType { get; }
125
126         /// <summary>
127         /// Gibt einen Wert zurück, der angibt, ob die Methode statisch ist.
128         /// </summary>
129         /// <value>
130         ///     <c>true</c>, wenn die Methode statisch ist, sonst <c>false</c>.
131         /// </value>
132         protected abstract bool IsMethodStatic { get; }
133
134         /// <summary>
135         /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl einer Methode.
```

```
136         /// </summary>
137         protected override void InitPorts()
138         {
139             foreach (var iP in inPortsByParam.Values)
140             {
141                 Children.Remove(iP);
142             }
143             foreach (var oP in outPortsByParam.Values)
144             {
145                 Children.Remove(oP);
146             }
147             inPortsByParam.Clear();
148             outPortsByParam.Clear();
149 
150             if (methodBase != null)
151             {
152                 int cnt = 0;
153                 foreach (var p in methodBase.GetParameters())
154                 {
155                     cnt++;
156                     if (p.IsIn || !p.IsOut)
157                     {
158                         var pType = p.ParameterType;
159 
160                         var iP = new InPort(this, "in_" + p.Name,
161                             pType.IsByRef ? pType.GetElementType() : pType);
162                         inPortsByParam.Add(p, iP);
163 
164                         if (pType.IsByRef)
165                         {
166                             var oP = new OutPort(this, "out_" + p.Name,
167                                 pType.GetElementType());
168                             outPortsByParam.Add(p, oP);
169                         }
170                     }
171                     else
172                     {
173                         var oP = new OutPort(this, string.Format("out_{0:00}", cnt),
174                             p.ParameterType.GetElementType());
175                         outPortsByParam.Add(p, oP);
176                     }
177                 }
178 
179                 if (!IsMethodStatic)
180                 {
181                     if (instanceIP == null)
182                     {
183                         instanceIP = new InPort(this, "instance",
184                             instanceTypeProvider);
185                     }
186                     instanceTypeProvider.SupportedDataType = methodBase.ReflectedType;
187                 }
188                 else
189                 {
190                     if (instanceIP != null)
191                     {
192                         Children.Remove(instanceIP);
193                         instanceIP = null;
194                     }
195                 }
196 
197                 var retType = ReturnType;
198                 if (retType != null && retType != typeof(void))
199                 {
200                     if (returnOP == null)
201                     {
202                         returnOP = new OutPort(this, ReturnOutPortId,
203                             returnTypeProvider);
204                     }
205                     returnTypeProvider.SupportedDataType = retType;
206                 }
207                 else
208                 {
209                     if (returnOP != null)
210                     {
211                         Children.Remove(returnOP);
212                         returnOP = null;
213                     }
214                 }
215 
216                 ClrTools.ConfigNodesParallelisation(this, MethodBase.ReflectedType);
217             }
218             else
219             {
220                 instanceTypeProvider.SupportedDataType = null;
221                 returnTypeProvider.SupportedDataType = null;
222             }
223 
```

```
224             UpdateVisualization();
225             Activate();
226         }
227 
228         /// <summary>
229         /// Wird aufgerufen, wenn der Benutzer den Standardbefehl des Knotens
230         /// aufgerufen hat.
231         /// </summary>
232         /// <remarks>
233         /// Diese Methode muss nicht überschrieben werden.
234         /// </remarks>
235         public override void OnDefaultCommand()
236         {
237             OnCommand("work");
238         }
239 
240         /// <summary>
241         /// Wird aufgerufen wenn der Benutzer einen Benutzerbefehl aufgerufen hat.
242         /// </summary>
243         /// <param name="command">Der Befehls-Code des Benutzerbefehls.</param>
244         /// <remarks>
245         /// Diese Methode sollte von einer Klasse überschrieben werden,
246         /// die einen oder mehrere Benutzerbefehle mit <see cref="Node.AddCommand"/>
247         /// registriert hat.
248         /// </remarks>
249         public override void OnCommand(string command)
250         {
251             if (command == "work")
252             {
253                 Activate();
254             }
255         }
256 
257         /// <summary>
258         /// Führt die Knoten-Operation durch.
259         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
260         /// </summary>
261         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
262         /// Tokens aller Eingänge.</param>
263         public override void Work(TokenSet ts)
264         {
265             if (methodBase == null)
266             {
267                 return;
268             }
269             if (triggerIP.IsConnected
270                 && (triggerIP.CurrentTokenState == TokenState.NotSet
271                     || !triggerIP.IsCurrentTokenNew))
272             {
273                 return;
274             }
275 
276             ts.ExcludeFromMinMaxCalculation(triggerIP.Name);
277             var minTS = ts.Count > 1 ? ts.MinTokenState : TokenState.Valid;
278 
279             var parameters = methodBase.GetParameters();
280             var parameterValues = new object[parameters.Length];
281             object instance = null;
282 
283             if (minTS > TokenState.NotSet)
284             {
285                 foreach (var kvp in inPortsByParam)
286                 {
287                     var t = ts[kvp.Value];
288                     parameterValues[kvp.Key.Position] = t.GetValue((object)null);
289                 }
290                 if (instanceIP != null)
291                 {
292                     instance = ts[instanceIP].Value;
293                     if (instance == null)
294                     {
295                         return;
296                     }
297                 }
298             }
299             else
300             {
301                 if (returnOP != null)
302                 {
303                     returnOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
304                 }
305                 triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
306                 return;
307             }
308             try
309             {
310                 object result;
311                 if (methodBase is ConstructorInfo)
```

```
312                 {
313                     var cI = (ConstructorInfo)methodBase;
314                     result = cI.Invoke(parameterValues);
315                 }
316                 else
317                 {
318                     TraceVerbose(0, "Versuche␣Methodenaufruf");
319                     result = methodBase.Invoke(instance, parameterValues);
320                 }
321                 if (returnOP != null)
322                 {
323                     returnOP.UpdateToken(result, minTS, ts.StreamOfSet, ts.IndexOfSet);
324                 }
325                 foreach (var kvp in outPortsByParam)
326                 {
327                     kvp.Value.UpdateToken(
328                         parameterValues[kvp.Key.Position],
329                         minTS, ts.StreamOfSet, ts.IndexOfSet);
330                 }
331                 triggerOP.UpdateToken(
332                     ControlTrigger.Instance, minTS,
333                     ts.StreamOfSet, ts.IndexOfSet);
334             }
335             catch (TargetInvocationException tie)
336             {
337                 if (returnOP != null)
338                 {
339                     returnOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
340                     foreach (var op in outPortsByParam.Values)
341                     {
342                         op.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
343                     }
344                 }
345                 triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
346                 TraceWarning(423, "Methodenaufruf␣fehlgeschlagen", tie);
347             }
348             catch (Exception e)
349             {
350                 TraceWarning(423, "Methodenaufruf␣fehlgeschlagen.", e);
351             }
352         }
353 
354         /// <summary>
355         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
356         /// welches das gekapselte .NET-Element repräsentieren.
357         /// </summary>
358         /// <returns>Das erste von zwei Symbolen.</returns>
359         protected override Image GetInteractionSymbol1()
360         {
361             return InteractionSymbols.Images["Method"];
362         }
363 
364         /// <summary>
365         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
366         /// welches das gekapselte .NET-Element repräsentieren.
367         /// </summary>
368         /// <returns>Das zweite von zwei Symbolen oder <c>null</c>.</returns>
369         protected override Image GetInteractionSymbol2()
370         {
371             return null;
372         }
373 
374         /// <summary>
375         /// Gibt die erste von drei möglichen Textzeilen zurück,
376         /// welche das gekapselte .NET-Element beschreiben.
377         /// </summary>
378         /// <returns>Die erste von drei Textzeilen.</returns>
379         protected override string GetTextLine1()
380         {
381             return MethodBase != null
382                        ? ClrTools.GetNamespaceCaption(MethodBase.ReflectedType.Namespace)
383                        : "";
384         }
385     }
386 }
```

Listing A.12: DynamicNode.Lib.Objects.AbstractMethod

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1476 $</revision>
```

```
9   // <author>$Author: Kiertscher.Tobias $</author>
10  // <date>$Date: 2010-04-26 15:56:29 +0200 (Mo, 26. Apr 2010) $</date>
11
12  #endregion
13
14  using System;
15  using System.Reflection;
16  using System.Windows.Forms;
17  using DynamicNode.Ext.Objects;
18  using DynamicNode.Ext.Visual;
19  using DynamicNode.Persistence;
20
21  namespace DynamicNode.Lib.Objects
22  {
23      public abstract class AbstractMethod : AbstractMethodBase
24      {
25          private static MethodSelector selector;
26
27          /// <summary>
28          /// Initialisiert eine neue Instanz von <see cref="NConstructor"/>.
29          /// </summary>
30          protected AbstractMethod()
31          {
32          }
33
34          /// <summary>
35          /// Initialisiert eine neue Instanz von <see cref="NConstructor"/> mit
36          /// einem vorgegebenen CLR-Constructor.
37          /// </summary>
38          /// <param name="methodInfo">Eine CLR-Methode die gekapselt werden soll.</param>
39          protected AbstractMethod(MethodInfo methodInfo)
40              : base(methodInfo)
41          {
42          }
43
44          /// <summary>
45          /// Gibt die Port-ID für den Ausgang zurück,
46          /// der Rückgabewerte des Methodenaufrufs weiterleiten soll.
47          /// </summary>
48          /// <value></value>
49          protected override string ReturnOutPortId
50          {
51              get { return "return"; }
52          }
53
54          /// <summary>
55          /// Gibt den Datentyp für den Ausgang zurück,
56          /// der den Rückgabewert des Methodenaufrufs weiterleiten soll.
57          /// </summary>
58          /// <value></value>
59          protected override Type ReturnType
60          {
61              get { return Method != null ? Method.ReturnType : null; }
62          }
63
64          protected MethodInfo Method
65          {
66              get { return (MethodInfo) MethodBase; }
67              set { MethodBase = value; }
68          }
69
70          /// <summary>
71          /// Initialisiert die Knoten-Steuerung.
72          /// </summary>
73          /// <param name="ctrlPanelExt">Die Knoten-Steuerung.</param>
74          protected override void InitControlPanel(NodeControlPanelExtension ctrlPanelExt)
75          {
76              if (selector == null)
77              {
78                  selector = new MethodSelector();
79                  selector.Dock = DockStyle.Fill;
80              }
81
82              ctrlPanelExt.Control = selector;
83              ctrlPanelExt.Show += ctrlPanelExt_Show;
84              ctrlPanelExt.Hide += ctrlPanelExt_Hide;
85          }
86
87          private void ctrlPanelExt_Show(object sender, EventArgs e)
88          {
89              InitSelector(selector);
90              selector.Method = Method;
91              selector.MethodSelected += Selector_MethodSelected;
92          }
93
94          private void ctrlPanelExt_Hide(object sender, EventArgs e)
95          {
96              selector.MethodSelected -= Selector_MethodSelected;
```

```
97          }
98
99          /// <summary>
100         /// Initialisiert den <see cref="MethodSelector"/>, der als Knotensteuerung dient.
101         /// </summary>
102         /// <param name="selector">Das Steuerelement zur Auswahl einer Methode.</param>
103         protected virtual void InitSelector(MethodSelector selector)
104         {
105         }
106
107         /// <summary>
108         /// Wird aufgerufen wenn der Graph gespeichert wird.
109         /// Wer diese Methode überschreibt muss <c>base.OnStore(pr)</c> aufrufen.
110         /// </summary>
111         /// <param name="pw">Der <see cref="IPropertyWriter"/> mit dem benutzerdefinierte
112         /// Eigenschaften des Knotens gespeichert werden können.</param>
113         public override void OnStore(IPropertyWriter pw)
114         {
115             base.OnStore(pw);
116             if (Method != null)
117             {
118                 pw.WriteProperty("method",
119                     "\n" + ClrTools.GetDescriptor(Method));
120             }
121         }
122
123         /// <summary>
124         /// Wird aufgerufen wenn der Graph rekonstruiert wird.
125         /// Wer diese Methode überschreibt muss <c>base.OnRestore(pr)</c> aufrufen.
126         /// In dieser Methode sollten die Eingänge wenn möglich mit Standard-Werten
127         /// belegt werden.
128         /// </summary>
129         /// <param name="pr">Der <see cref="IPropertyReader"/> mit dem benutzerdefinierte
130         /// Eigenschaften des Knotens gelesen werden können.</param>
131         /// <remarks>
132         /// Diese Methode wird beim Laden eines Graphen aufgerufen bevor die Verbindungen
133         /// zwischen den Knoten wiederhergstellt werden.
134         /// </remarks>
135         public override void OnRestore(IPropertyReader pr)
136         {
137             base.OnRestore(pr);
138             string descriptor = pr.ReadPropertyAsString("method", null);
139             if (descriptor != null)
140             {
141                 try
142                 {
143                     Method = ClrTools.FindMethod(descriptor);
144                     if (selector != null)
145                     {
146                         selector.Method = Method;
147                     }
148                 }
149                 catch (Exception e)
150                 {
151                     TraceWarning(429, "Eine␣Methode␣wurde␣nicht␣gefunden.", e);
152                 }
153             }
154         }
155
156         private void Selector_MethodSelected(object sender, EventArgs e)
157         {
158             Method = ((MethodSelector) sender).Method;
159         }
160
161         /// <summary>
162         /// Gibt die zweite von drei möglichen Textzeilen zurück,
163         /// welche das gekapselte .NET-Element beschreiben.
164         /// </summary>
165         /// <returns>Die zweite von drei Textzeilen.</returns>
166         protected override string GetTextLine2()
167         {
168             return MethodBase != null
169                 ? ClrTools.GetTypeCaption(MethodBase.ReflectedType)
170                     + "." + ClrTools.GetMethodCaption(Method, true)
171                 : "";
172         }
173     }
174 }
```

Listing A.13: DynamicNode.Lib.Objects.NMethod

```
1  #region file header
2
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
```

```
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1461 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-04-20 11:29:31 +0200 (Di, 20. Apr 2010) $</date>
11
12 #endregion
13
14 using System;
15 using System.Reflection;
16 using DynamicNode.Core;
17 using DynamicNode.Ext.Objects;
18
19 namespace DynamicNode.Lib.Objects
20 {
21     [ResourceNodeInfo("CLR")]
22     public class NMethod : AbstractMethod
23     {
24         public static NMethod Creator(MemberInfo memberInfo)
25         {
26             return new NMethod((MethodInfo)memberInfo);
27         }
28
29         /// <summary>
30         /// Initialisiert eine neue Instanz von <see cref="NMethod"/>.
31         /// </summary>
32         public NMethod() { }
33
34         /// <summary>
35         /// Initialisiert eine neue Instanz von <see cref="NMethod"/>
36         /// mit einer zu kapselden Methode.
37         /// </summary>
38         /// <param name="methodInfo">Die Methode.</param>
39         public NMethod(MethodInfo methodInfo) : base(methodInfo) { }
40
41         /// <summary>
42         /// Überprüft ob die übergebene Methode geeignet ist.
43         /// Falls ddie Methode ungeeignet ist wird eine <see cref="ArgumentException"/>
44         /// geworfen.
45         /// </summary>
46         /// <param name="mI">Die zu prüfende Methode.</param>
47         protected override void CheckMethodBase(MethodBase mI)
48         {
49             if (mI == null)
50             {
51                 throw new ArgumentNullException("mI");
52             }
53             if (!(mI is MethodInfo))
54             {
55                 throw new ArgumentException(
56                     "Es␣wurde␣ein␣Konstruktor␣an␣Stelle␣einer␣Methode␣übergeben.", "mI");
57             }
58             string msg;
59             if (!ClrTools.CheckMethod((MethodInfo)mI, false, out msg))
60             {
61                 throw new ArgumentException(msg, "mI");
62             }
63         }
64
65         /// <summary>
66         /// Gibt einen Wert zurück, der angibt, ob die Methode statisch ist.
67         /// </summary>
68         /// <value><c>true</c>, wenn die Methode statisch ist, sonst <c>false</c>.</value>
69         protected override bool IsMethodStatic { get { return false; } }
70
71         /// <summary>
72         /// Initialisiert den <see cref="MethodSelector"/>, der als Knotensteuerung dient.
73         /// </summary>
74         /// <param name="selector">Das Steuerelement zur Auswahl einer Methode.</param>
75         protected override void InitSelector(MethodSelector selector)
76         {
77             base.InitSelector(selector);
78             selector.MustBeStatic = false;
79         }
80     }
81 }
```

Listing A.14: DynamicNode.Lib.Objects.NStaticMethod

```
1 #region file header
2
3 // <copyright>
4 // Copyright (c) 2009-2010 by Tobias Kiertscher
5 // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6 // </copyright>
7 // <license>DNLicense.html</license>
```

```
8  // <revision>$Revision: 1490 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-04 21:29:52 +0200 (Di, 04. Mai 2010) $</date>
11
12 #endregion
13
14 using System;
15 using System.Reflection;
16 using DynamicNode.Core;
17 using DynamicNode.Ext.Objects;
18
19 namespace DynamicNode.Lib.Objects
20 {
21     [ResourceNodeInfo("CLR")]
22     public class NStaticMethod : AbstractMethod
23     {
24         public static NStaticMethod Creator(MemberInfo memberInfo)
25         {
26             return new NStaticMethod((MethodInfo)memberInfo);
27         }
28
29         /// <summary>
30         /// Initialisiert eine neue Instanz von <see cref="NStaticMethod"/>.
31         /// </summary>
32         public NStaticMethod() { }
33
34         /// <summary>
35         /// Initialisiert eine neue Instanz von <see cref="NStaticMethod"/>
36         /// mit einer zu kapselnden statischen Methode.
37         /// </summary>
38         /// <param name="methodInfo">Die statische Methode.</param>
39         public NStaticMethod(MethodInfo methodInfo) : base(methodInfo) { }
40
41         /// <summary>
42         /// Überprüft ob die übergebene Methode geeignet ist.
43         /// Falls ddie Methode ungeeignet ist wird eine <see cref="ArgumentException"/>
44         /// geworfen.
45         /// </summary>
46         /// <param name="mI">Die zu prüfende Methode.</param>
47         protected override void CheckMethodBase(MethodBase mI)
48         {
49             if (mI == null)
50             {
51                 throw new ArgumentNullException("mI");
52             }
53             if (!(mI is MethodInfo))
54             {
55                 throw new ArgumentException(
56                     "Es␣wurde␣ein␣Konstruktor␣an␣Stelle␣einer␣Methode␣übergeben.", "mI");
57             }
58             string msg;
59             if (!ClrTools.CheckMethod((MethodInfo)mI, true, out msg))
60             {
61                 throw new ArgumentException(msg, "mI");
62             }
63         }
64
65         /// <summary>
66         /// Gibt einen Wert zurück, der angibt, ob die Methode statisch ist.
67         /// </summary>
68         /// <value><c>true</c>, wenn die Methode statisch ist, sonst <c>false</c>.</value>
69         protected override bool IsMethodStatic { get { return true; } }
70
71         /// <summary>
72         /// Initialisiert den <see cref="MethodSelector"/>, der als Knotensteuerung dient.
73         /// </summary>
74         /// <param name="selector">Das Steuerelement zur Auswahl einer Methode.</param>
75         protected override void InitSelector(MethodSelector selector)
76         {
77             base.InitSelector(selector);
78             selector.MustBeStatic = true;
79         }
80
81         /// <summary>
82         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
83         /// welches das gekapselte .NET-Element repräsentieren.
84         /// </summary>
85         /// <returns>Das erste von zwei Symbolen.</returns>
86         protected override System.Drawing.Image GetInteractionSymbol1()
87         {
88             return InteractionSymbols.Images["Method_Static"];
89         }
90     }
91 }
```

Listing A.15: DynamicNode.Lib.Objects.NConstructor

```
#region file header

// <copyright>
// Copyright (c) 2009-2010 by Tobias Kiertscher
// Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
// </copyright>
// <license>DNLicense.html</license>
// <revision>$Revision: 1490 $</revision>
// <author>$Author: Kiertscher.Tobias $</author>
// <date>$Date: 2010-05-04 21:29:52 +0200 (Di, 04. Mai 2010) $</date>

#endregion

using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using DynamicNode.Core;
using DynamicNode.Ext.Objects;
using DynamicNode.Ext.Visual;
using DynamicNode.Persistence;

namespace DynamicNode.Lib.Objects
{
    [ResourceNodeInfo("CLR")]
    public class NConstructor : AbstractMethodBase
    {
        public static NConstructor Creator(MemberInfo memberInfo)
        {
            return new NConstructor((ConstructorInfo)memberInfo);
        }

        private static ConstructorSelector selector;

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="NConstructor"/>.
        /// </summary>
        public NConstructor() { }

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="NConstructor"/> mit
        /// einem vorgegebenen CLR-Constructor.
        /// </summary>
        /// <param name="constructorInfo">Ein CLR-Constructor der gekapselt werden soll.</
            param>
        public NConstructor(ConstructorInfo constructorInfo) : base(constructorInfo) { }

        /// <summary>
        /// Überprüft ob die übergebene Methode geeignet ist.
        /// Falls ddie Methode ungeeignet ist wird eine <see cref="ArgumentException"/>
        /// geworfen.
        /// </summary>
        /// <param name="mI">Die zu prüfende Methode.</param>
        protected override void CheckMethodBase(MethodBase mI)
        {
            if (mI == null)
            {
                throw new ArgumentNullException("mI");
            }
            if (!(mI is ConstructorInfo))
            {
                throw new ArgumentException("Die␣übergebene␣Methode␣ist␣kein␣Constructor.",
                    "mI");
            }
            string msg;
            if (!ClrTools.CheckConstructor((ConstructorInfo)mI, out msg))
            {
                throw new ArgumentException(msg, "mI");
            }
        }

        /// <summary>
        /// Gibt die Port-ID für den Ausgang zurück,
        /// der Rückgabewerte des Methodenaufrufs weiterleiten soll.
        /// </summary>
        /// <value></value>
        protected override string ReturnOutPortId { get { return "instance"; } }

        /// <summary>
        /// Gibt den Datentyp für den Ausgang zurück,
        /// der den Rückgabewert des Methodenaufrufs weiterleiten soll.
        /// </summary>
        /// <value></value>
        protected override Type ReturnType
        {
            get { return Constructor != null ? Constructor.ReflectedType : null; }
```

```
85          }
86
87          /// <summary>
88          /// Gibt einen Wert zurück, der angibt, ob die Methode statisch ist.
89          /// </summary>
90          /// <value><c>true</c>, wenn die Methode statisch ist, sonst <c>false</c>.</value>
91          protected override bool IsMethodStatic
92          {
93              get { return true; }
94          }
95
96          /// <summary>
97          /// Gibt das Metadaten-Objekt für den gekapselten Konstruktor zurück.
98          /// </summary>
99          /// <value>Die Metadaten des Konstruktors.</value>
100         private ConstructorInfo Constructor
101         {
102             get { return (ConstructorInfo)MethodBase; }
103             set { MethodBase = value; }
104         }
105
106         /// <summary>
107         /// Initialisiert die Knoten-Steuerung.
108         /// </summary>
109         /// <param name="ctrlPanelExt">Die Knoten-Steuerung.</param>
110         protected override void InitControlPanel(NodeControlPanelExtension ctrlPanelExt)
111         {
112             if (selector == null)
113             {
114                 selector = new ConstructorSelector();
115                 selector.Dock = DockStyle.Fill;
116             }
117
118             ctrlPanelExt.Control = selector;
119             ctrlPanelExt.Show += ctrlPanelExt_Show;
120             ctrlPanelExt.Hide += ctrlPanelExt_Hide;
121         }
122
123         private void ctrlPanelExt_Hide(object sender, EventArgs e)
124         {
125             selector.ConstructorSelected -= ConstructorSelector_ConstructorSelected;
126         }
127
128         private void ctrlPanelExt_Show(object sender, EventArgs e)
129         {
130             selector.Constructor = Constructor;
131             selector.ConstructorSelected += ConstructorSelector_ConstructorSelected;
132         }
133
134         private void ConstructorSelector_ConstructorSelected(object sender, EventArgs e)
135         {
136             Constructor = ((ConstructorSelector)sender).Constructor;
137         }
138
139         /// <summary>
140         /// Wird aufgerufen wenn der Graph gespeichert wird.
141         /// Wer diese Methode überschreibt muss <c>base.OnStore(pr)</c> aufrufen.
142         /// </summary>
143         /// <param name="pw">Der <see cref="IPropertyWriter"/> mit dem benutzerdefinierte
144         /// Eigenschaften des Knotens gespeichert werden können.</param>
145         public override void OnStore(IPropertyWriter pw)
146         {
147             base.OnStore(pw);
148             if (Constructor != null)
149             {
150                 pw.WriteProperty("constructor",
151                     "\n" + ClrTools.GetDescriptor(Constructor));
152             }
153         }
154
155         /// <summary>
156         /// Wird aufgerufen wenn der Graph rekonstruiert wird.
157         /// Wer diese Methode überschreibt muss <c>base.OnRestore(pr)</c> aufrufen.
158         /// In dieser Methode sollten die Eingänge wenn möglich mit Standard-Werten
159         /// belegt werden.
160         /// </summary>
161         /// <param name="pr">Der <see cref="IPropertyReader"/> mit dem benutzerdefinierte
162         /// Eigenschaften des Knotens gelesen werden können.</param>
163         /// <remarks>
164         /// Diese Methode wird beim Laden eines Graphen aufgerufen bevor die Verbindungen
165         /// zwischen den Knoten wiederhergstellt werden.
166         /// </remarks>
167         public override void OnRestore(IPropertyReader pr)
168         {
169             base.OnRestore(pr);
170             string descriptor = pr.ReadPropertyAsString("constructor", null);
171             if (descriptor != null)
172             {
```

```
173                 try
174                 {
175                     Constructor = ClrTools.FindConstructor(descriptor);
176                     if (selector != null)
177                     {
178                         selector.Constructor = Constructor;
179                     }
180                 }
181                 catch (Exception e)
182                 {
183                     TraceWarning(432, "Ein␣Constructor␣wurde␣nicht␣gefunden.", e);
184                 }
185             }
186         }
187 
188         /// <summary>
189         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
190         /// welches das gekapselte .NET-Element repräsentieren.
191         /// </summary>
192         /// <returns>Das erste von zwei Symbolen.</returns>
193         protected override Image GetInteractionSymbol1()
194         {
195             return InteractionSymbols.Images["Constructor"];
196         }
197 
198         /// <summary>
199         /// Gibt die zweite von drei möglichen Textzeilen zurück,
200         /// welche das gekapselte .NET-Element beschreiben.
201         /// </summary>
202         /// <returns>Die zweite von drei Textzeilen.</returns>
203         protected override string GetTextLine2()
204         {
205             return MethodBase != null
206                 ? "new␣" + ClrTools.GetTypeCaption(MethodBase.ReflectedType)
207                 : "";
208         }
209     }
210 }
```

Listing A.16: DynamicNode.Lib.Objects.AbstractField

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1476 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-04-26 15:56:29 +0200 (Mo, 26. Apr 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.Drawing;
16 using System.Reflection;
17 using System.Windows.Forms;
18 using DynamicNode.Ext.Objects;
19 using DynamicNode.Ext.Visual;
20 using DynamicNode.Persistence;
21 
22 namespace DynamicNode.Lib.Objects
23 {
24     public abstract class AbstractField : AbstractWrapper
25     {
26         private static FieldSelector selector;
27 
28         protected FieldInfo Field { get; private set; }
29 
30         /// <summary>
31         /// Initialisiert eine neue Instanz von <see cref="AbstractProperty"/>.
32         /// </summary>
33         protected AbstractField() {}
34 
35         /// <summary>
36         /// Initialisiert eine neue Instanz von <see cref="AbstractProperty"/>.
37         /// </summary>
38         /// <param name="fieldInfo">Ein zu kapselndes CLR-Field.</param>
39         protected AbstractField(FieldInfo fieldInfo)
40         {
41             CheckFieldInfo(fieldInfo);
42             Field = fieldInfo;
43             InitPorts();
44         }
45 
```

```
46          /// <summary>
47          /// Überprüft ob das übergebene Property geeignet ist.
48          /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
49          /// geworfen.
50          /// </summary>
51          /// <param name="fI">Das zu prüfende Property.</param>
52          protected abstract void CheckFieldInfo(FieldInfo fI);
53
54          /// <summary>
55          /// Initialisiert den <see cref="PropertySelector"/>, der als
56          /// Knotensteuerung dient.
57          /// </summary>
58          /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
59          protected abstract void InitSelector(FieldSelector selector);
60
61          /// <summary>
62          /// Initialisiert die Knoten-Steuerung.
63          /// </summary>
64          /// <param name="ctrlPanelExt">Die Knoten-Steuerung.</param>
65          protected override void InitControlPanel(NodeControlPanelExtension ctrlPanelExt)
66          {
67              if (selector == null)
68              {
69                  selector = new FieldSelector();
70                  selector.Dock = DockStyle.Fill;
71              }
72              ctrlPanelExt.Control = selector;
73              ctrlPanelExt.Show += ctrlPanelExt_Show;
74              ctrlPanelExt.Hide += ctrlPanelExt_Hide;
75          }
76
77          private void ctrlPanelExt_Show(object sender, EventArgs e)
78          {
79              InitSelector(selector);
80              selector.Field = Field;
81              selector.FieldSelected += Selector_FieldSelected;
82          }
83
84          private void ctrlPanelExt_Hide(object sender, EventArgs e)
85          {
86              selector.FieldSelected -= Selector_FieldSelected;
87          }
88
89          private void Selector_FieldSelected(object sender, EventArgs e)
90          {
91              var f = ((FieldSelector) sender).Field;
92              try
93              {
94                  CheckFieldInfo(f);
95              }
96              catch (Exception exc)
97              {
98                  TraceWarning(427, "Ungültiges␣Field␣ausgewählt.", exc);
99                  f = null;
100             }
101
102             Field = f;
103             InitPorts();
104         }
105
106         /// <summary>
107         /// Wird aufgerufen wenn der Graph gespeichert wird.
108         /// Wer diese Methode überschreibt muss <c>base.OnStore(pr)</c> aufrufen.
109         /// </summary>
110         /// <param name="pw">Der <see cref="IPropertyWriter"/> mit dem benutzerdefinierte
111         /// Eigenschaften des Knotens gespeichert werden können.</param>
112         public override void OnStore(IPropertyWriter pw)
113         {
114             base.OnStore(pw);
115             if (Field != null)
116             {
117                 pw.WriteProperty("field",
118                     "\n" + ClrTools.GetDescriptor(Field));
119             }
120         }
121
122         /// <summary>
123         /// Wird aufgerufen wenn der Graph rekonstruiert wird.
124         /// Wer diese Methode überschreibt muss <c>base.OnRestore(pr)</c> aufrufen.
125         /// In dieser Methode sollten die Eingänge wenn möglich mit Standard-Werten
126         /// belegt werden.
127         /// </summary>
128         /// <param name="pr">Der <see cref="IPropertyReader"/> mit dem benutzerdefinierte
129         /// Eigenschaften des Knotens gelesen werden können.</param>
130         /// <remarks>
131         /// Diese Methode wird beim Laden eines Graphen aufgerufen bevor die Verbindungen
132         /// zwischen den Knoten wiederhergstellt werden.
133         /// </remarks>
```

```
134         public override void OnRestore(IPropertyReader pr)
135         {
136             base.OnRestore(pr);
137             string descriptor = pr.ReadPropertyAsString("field", null);
138             if (descriptor != null)
139             {
140                 try
141                 {
142                     Field = ClrTools.FindField(descriptor);
143                     InitPorts();
144                     if (selector != null)
145                     {
146                         selector.Field = Field;
147                     }
148                 }
149                 catch (Exception e)
150                 {
151                     TraceWarning(428, "Ein␣Field␣wurde␣nicht␣gefunden.", e);
152                 }
153             }
154         }
155 
156         /// <summary>
157         /// Gibt die erste von drei möglichen Textzeilen zurück,
158         /// welche das gekapselte .NET-Element beschreiben.
159         /// </summary>
160         /// <returns>Die erste von drei Textzeilen.</returns>
161         protected override string GetTextLine1()
162         {
163             return Field != null
164                 ? ClrTools.GetNamespaceCaption(Field.ReflectedType.Namespace)
165                 : "";
166         }
167 
168         /// <summary>
169         /// Gibt die zweite von drei möglichen Textzeilen zurück,
170         /// welche das gekapselte .NET-Element beschreiben.
171         /// </summary>
172         /// <returns>Die zweite von drei Textzeilen.</returns>
173         protected override string GetTextLine2()
174         {
175             return Field != null
176                 ? ClrTools.GetTypeCaption(Field.ReflectedType)
177                     + "." + ClrTools.GetFieldCaption(Field)
178                 : "";
179         }
180 
181         /// <summary>
182         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
183         /// welches das gekapselte .NET-Element repräsentieren.
184         /// </summary>
185         /// <returns>Das erste von zwei Symbolen.</returns>
186         protected override Image GetInteractionSymbol1()
187         {
188             return InteractionSymbols.Images["Field"];
189         }
190     }
191 }
```

Listing A.17: DynamicNode.Lib.Objects.NRead

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1537 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.ComponentModel;
16 using System.Drawing;
17 using System.Reflection;
18 using DynamicNode.Core;
19 using DynamicNode.Ext.Objects;
20 
21 namespace DynamicNode.Lib.Objects
22 {
23     [ResourceNodeInfo("CLR")]
24     public class NRead : AbstractField
25     {
```

```
26          public static NRead Creator(MemberInfo memberInfo)
27          {
28              return new NRead((FieldInfo)memberInfo);
29          }
30
31          private InPortSingleDataTypeProvider instanceTypeProvider;
32          private InPort instanceIP;
33          private InPort triggerIP;
34          private OutPortDataTypeProvider valueTypeProvider;
35          private OutPort valueOP;
36          private OutPort triggerOP;
37
38          /// <summary>
39          /// Initialisiert eine neue Instanz von <see cref="NRead"/>.
40          /// </summary>
41          public NRead() { }
42
43          /// <summary>
44          /// Initialisiert eine neue Instanz von <see cref="NRead"/> mit
45          /// einem vorgegebenen CLR-Feld.
46          /// </summary>
47          /// <param name="fieldInfo">Ein CLR-Feld das gekapselt werden soll.</param>
48          public NRead(FieldInfo fieldInfo) : base(fieldInfo) { }
49
50          /// <summary>
51          /// Initialisiert die aktuelle Instanz.
52          /// Wird von den Konstruktoren aufgerufen.
53          /// </summary>
54          protected override void Init()
55          {
56              base.Init();
57              AttachExtension(instanceTypeProvider = new InPortSingleDataTypeProvider());
58              instanceIP = new InPort(this, "instance", instanceTypeProvider);
59              triggerIP = new InPort(this, "triggerIn");
60              AttachExtension(valueTypeProvider = new OutPortDataTypeProvider());
61              valueOP = new OutPort(this, "value", valueTypeProvider);
62              triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
63          }
64
65          /// <summary>
66          /// Überprüft ob das übergebene Property geeignet ist.
67          /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
68          /// geworfen.
69          /// </summary>
70          /// <param name="fI">Das zu prüfende Feld.</param>
71          protected override void CheckFieldInfo(FieldInfo fI)
72          {
73              if (fI == null)
74              {
75                  throw new ArgumentNullException("fI");
76              }
77              string msg;
78              if (!ClrTools.CheckField(fI, false, false, out msg))
79              {
80                  throw new ArgumentException(msg, "fI");
81              }
82          }
83
84          /// <summary>
85          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
86          /// </summary>
87          protected override void InitPorts()
88          {
89              if (Field != null)
90              {
91                  instanceTypeProvider.SupportedDataType = Field.ReflectedType;
92                  valueTypeProvider.SupportedDataType = Field.FieldType;
93
94                  ClrTools.ConfigNodesParallelisation(this, Field.ReflectedType);
95              }
96              else
97              {
98                  instanceTypeProvider.SupportedDataType = null;
99                  valueTypeProvider.SupportedDataType = null;
100             }
101
102             UpdateVisualization();
103             //Activate();
104         }
105
106         /// <summary>
107         /// Initialisiert den <see cref="FieldSelector"/>, der als
108         /// Knotensteuerung dient.
109         /// </summary>
110         /// <param name="selector">Das Steuerelement zur Auswahl eines Feldes.</param>
111         protected override void InitSelector(FieldSelector selector)
112         {
113             selector.MustBeStatic = false;
```

```
114             selector.Writeable = false;
115         }
116 
117         /// <summary>
118         /// Führt die Knoten-Operation durch.
119         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
120         /// </summary>
121         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
122         /// Tokens aller Eingänge.</param>
123         public override void Work(TokenSet ts)
124         {
125             if (Field == null)
126             {
127                 return;
128             }
129             if (triggerIP.IsConnected
130                 && (triggerIP.CurrentTokenState == TokenState.NotSet
131                     || !triggerIP.IsCurrentTokenNew))
132             {
133                 return;
134             }
135 
136             ts.ExcludeFromMinMaxCalculation(triggerIP.Name);
137             var minTS = ts.MinTokenState;
138 
139             if (minTS != TokenState.NotSet && ts[instanceIP].Value != null)
140             {
141                 var instance = ts[instanceIP].Value;
142                 try
143                 {
144                     var value = Field.GetValue(instance);
145 
146                     valueOP.UpdateToken(value, minTS, ts.StreamOfSet, ts.IndexOfSet);
147                     triggerOP.UpdateToken(
148                         ControlTrigger.Instance,
149                         minTS, ts.StreamOfSet, ts.IndexOfSet);
150                     return;
151                 }
152                 catch (TargetInvocationException)
153                 {
154                     valueOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
155                     return;
156                 }
157                 catch (Exception e)
158                 {
159                     TraceWarning(439, "Lesen␣des␣Feldes␣fehlgeschlagen.", e);
160                     return;
161                 }
162             }
163             valueOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
164             triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
165         }
166 
167         /// <summary>
168         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
169         /// welches das gekapselte .NET-Element repräsentieren.
170         /// </summary>
171         /// <returns>Das zweite von zwei Symbolen oder <c>null</c>.</returns>
172         protected override Image GetInteractionSymbol2()
173         {
174             return Field == null || Field.IsInitOnly
175                        ? null
176                        : InteractionSymbols.Images["Pull"];
177         }
178     }
179 }
```

Listing A.18: DynamicNode.Lib.Objects.NWrite

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1537 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.ComponentModel;
16 using System.Drawing;
17 using System.Reflection;
```

```
18  using DynamicNode.Core;
19  using DynamicNode.Ext.Objects;
20
21  namespace DynamicNode.Lib.Objects
22  {
23      [ResourceNodeInfo("CLR")]
24      public class NWrite : AbstractField
25      {
26          public static NWrite Creator(MemberInfo memberInfo)
27          {
28              return new NWrite((FieldInfo)memberInfo);
29          }
30
31          private InPortSingleDataTypeProvider instanceTypeProvider;
32          private InPort instanceIP;
33          private InPortSingleDataTypeProvider valueTypeProvider;
34          private InPort valueIP;
35          private InPort triggerIP;
36          private OutPort triggerOP;
37
38          /// <summary>
39          /// Initialisiert eine neue Instanz von <see cref="NWrite"/>.
40          /// </summary>
41          public NWrite() { }
42
43          /// <summary>
44          /// Initialisiert eine neue Instanz von <see cref="NWrite"/> mit
45          /// einem vorgegebenen CLR-Feld.
46          /// </summary>
47          /// <param name="fieldInfo">Ein CLR-Feld das gekapselt werden soll.</param>
48          public NWrite(FieldInfo fieldInfo) : base(fieldInfo) { }
49
50          /// <summary>
51          /// Initialisiert die aktuelle Instanz.
52          /// Wird von den Konstruktoren aufgerufen.
53          /// </summary>
54          protected override void Init()
55          {
56              base.Init();
57              AttachExtension(instanceTypeProvider = new InPortSingleDataTypeProvider());
58              instanceIP = new InPort(this, "instance", instanceTypeProvider);
59              AttachExtension(valueTypeProvider = new InPortSingleDataTypeProvider());
60              valueIP = new InPort(this, "value", valueTypeProvider);
61              triggerIP = new InPort(this, "triggerIn");
62              triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
63          }
64
65          /// <summary>
66          /// Überprüft ob das übergebene Field geeignet ist.
67          /// Falls das Field ungeeignet ist wird eine <see cref="ArgumentException"/>
68          /// geworfen.
69          /// </summary>
70          /// <param name="fI">Das zu prüfende Field.</param>
71          protected override void CheckFieldInfo(FieldInfo fI)
72          {
73              if (fI == null)
74              {
75                  throw new ArgumentNullException("fI");
76              }
77              string msg;
78              if (!ClrTools.CheckField(fI, false, true, out msg))
79              {
80                  throw new ArgumentException(msg, "fI");
81              }
82          }
83
84          /// <summary>
85          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
86          /// </summary>
87          protected override void InitPorts()
88          {
89              if (Field != null)
90              {
91                  instanceTypeProvider.SupportedDataType = Field.ReflectedType;
92                  valueTypeProvider.SupportedDataType = Field.FieldType;
93
94                  ClrTools.ConfigNodesParallelisation(this, Field.ReflectedType);
95              }
96              else
97              {
98                  instanceTypeProvider.SupportedDataType = null;
99                  valueTypeProvider.SupportedDataType = null;
100             }
101
102             UpdateVisualization();
103             //Activate();
104         }
105
```

```
106         /// <summary>
107         /// Initialisiert den <see cref="FieldSelector"/>, der als
108         /// Knotensteuerung dient.
109         /// </summary>
110         /// <param name="selector">Das Steuerelement zur Auswahl eines Fields.</param>
111         protected override void InitSelector(FieldSelector selector)
112         {
113             selector.MustBeStatic = false;
114             selector.Writeable = true;
115         }
116 
117         /// <summary>
118         /// Führt die Knoten-Operation durch.
119         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
120         /// </summary>
121         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
122         /// Tokens aller Eingänge.</param>
123         public override void Work(TokenSet ts)
124         {
125             if (Field == null)
126             {
127                 return;
128             }
129             if (triggerIP.IsConnected
130                 && (triggerIP.CurrentTokenState == TokenState.NotSet
131                     || !triggerIP.IsCurrentTokenNew))
132             {
133                 return;
134             }
135 
136             ts.ExcludeFromMinMaxCalculation(triggerIP.Name);
137             var minTS = ts.MinTokenState;
138             if (minTS != TokenState.NotSet && ts[instanceIP].Value != null)
139             {
140                 var instance = ts[instanceIP].Value;
141                 var value = ts[valueIP].Value;
142                 try
143                 {
144                     Field.SetValue(instance, value);
145 
146                     triggerOP.UpdateToken(
147                         ControlTrigger.Instance,
148                         minTS, ts.StreamOfSet, ts.IndexOfSet);
149                     return;
150                 }
151                 catch (TargetInvocationException)
152                 {
153                     return;
154                 }
155                 catch (Exception e)
156                 {
157                     TraceWarning(445, "Schreiben␣des␣Feldes␣fehlgeschlagen.", e);
158                     return;
159                 }
160             }
161             triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
162         }
163 
164         /// <summary>
165         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
166         /// welches das gekapselte .NET-Element repräsentieren.
167         /// </summary>
168         /// <returns>
169         /// Das zweite von zwei Symbolen oder <c>null</c>.
170         /// </returns>
171         protected override Image GetInteractionSymbol2()
172         {
173             return InteractionSymbols.Images["Push"];
174         }
175     }
176 }
```

Listing A.19: DynamicNode.Lib.Objects.NStaticRead

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1537 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>
11 
12 #endregion
```

```
13
14  using System;
15  using System.Drawing;
16  using System.Reflection;
17  using DynamicNode.Core;
18  using DynamicNode.Ext.Objects;
19
20  namespace DynamicNode.Lib.Objects
21  {
22      [ResourceNodeInfo("CLR")]
23      public class NStaticRead : AbstractField
24      {
25          public static NStaticRead Creator(MemberInfo memberInfo)
26          {
27              return new NStaticRead((FieldInfo) memberInfo);
28          }
29
30          private InPort triggerIP;
31          private OutPortDataTypeProvider valueTypeProvider;
32          private OutPort valueOP;
33          private OutPort triggerOP;
34
35          /// <summary>
36          /// Initialisiert eine neue Instanz von <see cref="NStaticRead"/>.
37          /// </summary>
38          public NStaticRead(){}
39
40          /// <summary>
41          /// Initialisiert eine neue Instanz von <see cref="NStaticRead"/> mit
42          /// einem vorgegebenen CLR-Field.
43          /// </summary>
44          /// <param name="fieldInfo">Ein CLR-Field das gekapselt werden soll.</param>
45          public NStaticRead(FieldInfo fieldInfo): base(fieldInfo){}
46
47          /// <summary>
48          /// Initialisiert die aktuelle Instanz.
49          /// Wird von den Konstruktoren aufgerufen.
50          /// </summary>
51          protected override void Init()
52          {
53              base.Init();
54              triggerIP = new InPort(this, "triggerIn");
55              AttachExtension(valueTypeProvider = new OutPortDataTypeProvider());
56              valueOP = new OutPort(this, "value", valueTypeProvider);
57              triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
58          }
59
60          /// <summary>
61          /// Überprüft ob das übergebene field geeignet ist.
62          /// Falls das field ungeeignet ist wird eine <see cref="ArgumentException"/>
63          /// geworfen.
64          /// </summary>
65          /// <param name="fI">Das zu prüfende field.</param>
66          protected override void CheckFieldInfo(FieldInfo fI)
67          {
68              if (fI == null)
69              {
70                  throw new ArgumentNullException("fI");
71              }
72              string msg;
73              if (!ClrTools.CheckField(fI, true, false, out msg))
74              {
75                  throw new ArgumentException(msg, "fI");
76              }
77          }
78
79          /// <summary>
80          /// Initialisiert den <see cref="FieldSelector"/>, der als
81          /// Knotensteuerung dient.
82          /// </summary>
83          /// <param name="selector">Das Steuerelement zur Auswahl eines Fields.</param>
84          protected override void InitSelector(FieldSelector selector)
85          {
86              selector.MustBeStatic = true;
87              selector.Writeable = false;
88          }
89
90          /// <summary>
91          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
92          /// </summary>
93          protected override void InitPorts()
94          {
95              valueTypeProvider.SupportedDataType =
96                  Field != null
97                      ? Field.FieldType
98                      : null;
99
100             UpdateVisualization();
```

```
101             Activate();
102         }
103 
104         /// <summary>
105         /// Führt die Knoten-Operation durch.
106         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
107         /// </summary>
108         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
109         /// Tokens aller Eingänge.</param>
110         public override void Work(TokenSet ts)
111         {
112             if (Field == null)
113             {
114                 return;
115             }
116             if (triggerIP.IsConnected
117                 && (triggerIP.CurrentTokenState == TokenState.NotSet
118                     || !triggerIP.IsCurrentTokenNew))
119             {
120                 return;
121             }
122 
123             try
124             {
125                 var value = Field.GetValue(null);
126 
127                 valueOP.UpdateToken(value, TokenState.Valid, ts.StreamOfSet, ts.IndexOfSet)
                        ;
128                 triggerOP.UpdateToken(
129                     ControlTrigger.Instance,
130                     TokenState.Valid, ts.StreamOfSet, ts.IndexOfSet);
131                 return;
132             }
133             catch (TargetInvocationException)
134             {
135                 valueOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
136                 return;
137             }
138             catch (Exception e)
139             {
140                 TraceWarning(442, "Lesen␣des␣statischen␣Feldes␣fehlgeschlagen.", e);
141                 return;
142             }
143         }
144 
145         /// <summary>
146         /// Gets the interaction symbol1.
147         /// </summary>
148         /// <returns></returns>
149         protected override Image GetInteractionSymbol1()
150         {
151             if (Field != null)
152             {
153                 return InteractionSymbols.Images[
154                     Field.ReflectedType.IsEnum && !Field.Name.Equals("value__")
155                         ? "EnumItem"
156                         : Field.IsLiteral
157                               ? "Constant"
158                               : "Field_Static"];
159             }
160             return InteractionSymbols.Images["Field_Static"];
161         }
162 
163         /// <summary>
164         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
165         /// welches das gekapselte .NET-Element repräsentieren.
166         /// </summary>
167         /// <returns>
168         /// Das zweite von zwei Symbolen oder <c>null</c>.
169         /// </returns>
170         protected override Image GetInteractionSymbol2()
171         {
172             if (Field != null)
173             {
174                 return (Field.ReflectedType.IsEnum && !Field.Name.Equals("value__"))
175                     || Field.IsLiteral
176 //                    || Field.IsInitOnly
177                       ? null
178                       : InteractionSymbols.Images["Pull"];
179             }
180             return null;
181         }
182     }
183 }
```

Listing A.20: DynamicNode.Lib.Objects.NStaticWrite

```
#region file header

// <copyright>
// Copyright (c) 2009-2010 by Tobias Kiertscher
// Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
// </copyright>
// <license>DNLicense.html</license>
// <revision>$Revision: 1537 $</revision>
// <author>$Author: Kiertscher.Tobias $</author>
// <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>

#endregion

using System;
using System.Drawing;
using System.Reflection;
using DynamicNode.Core;
using DynamicNode.Ext.Objects;

namespace DynamicNode.Lib.Objects
{
    [ResourceNodeInfo("CLR")]
    public class NStaticWrite : AbstractField
    {
        public static NStaticWrite Creator(MemberInfo memberInfo)
        {
            return new NStaticWrite((FieldInfo)memberInfo);
        }

        private InPortSingleDataTypeProvider valueTypeProvider;
        private InPort valueIP;
        private InPort triggerIP;
        private OutPort triggerOP;

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="NStaticWrite"/>.
        /// </summary>
        public NStaticWrite() { }

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="NStaticWrite"/> mit
        /// einem vorgegebenen CLR-Field.
        /// </summary>
        /// <param name="fieldInfo">Ein CLR-Field das gekapselt werden soll.</param>
        public NStaticWrite(FieldInfo fieldInfo) : base(fieldInfo) { }

        /// <summary>
        /// Initialisiert die aktuelle Instanz.
        /// Wird von den Konstruktoren aufgerufen.
        /// </summary>
        protected override void Init()
        {
            base.Init();
            AttachExtension(valueTypeProvider = new InPortSingleDataTypeProvider());
            valueIP = new InPort(this, "value", valueTypeProvider);
            triggerIP = new InPort(this, "triggerIn");
            triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
        }

        /// <summary>
        /// Überprüft ob das übergebene Field geeignet ist.
        /// Falls das Field ungeeignet ist wird eine <see cref="ArgumentException"/>
        /// geworfen.
        /// </summary>
        /// <param name="fI">Das zu prüfende Field.</param>
        protected override void CheckFieldInfo(FieldInfo fI)
        {
            if (fI == null)
            {
                throw new ArgumentNullException("fI");
            }
            string msg;
            if (!ClrTools.CheckField(fI, true, true, out msg))
            {
                throw new ArgumentException(msg, "fI");
            }
        }

        /// <summary>
        /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
        /// </summary>
        protected override void InitPorts()
        {
            valueTypeProvider.SupportedDataType =
                Field != null
                    ? Field.FieldType
```

```
87                      : null;
88
89              UpdateVisualization();
90              //Activate();
91          }
92
93          /// <summary>
94          /// Initialisiert den <see cref="FieldSelector"/>, der als
95          /// Knotensteuerung dient.
96          /// </summary>
97          /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
98          protected override void InitSelector(FieldSelector selector)
99          {
100             selector.MustBeStatic = true;
101             selector.Writeable = true;
102         }
103
104         /// <summary>
105         /// Führt die Knoten-Operation durch.
106         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
107         /// </summary>
108         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
109         /// Tokens aller Eingänge.</param>
110         public override void Work(TokenSet ts)
111         {
112             if (Field == null)
113             {
114                 return;
115             }
116             if (triggerIP.IsConnected
117                 && (triggerIP.CurrentTokenState == TokenState.NotSet
118                     || !triggerIP.IsCurrentTokenNew))
119             {
120                 return;
121             }
122
123             var vt = ts[valueIP];
124             if (vt.State != TokenState.NotSet)
125             {
126                 try
127                 {
128                     Field.SetValue(null, vt.Value);
129
130                     triggerOP.UpdateToken(
131                         ControlTrigger.Instance,
132                         vt.State, ts.StreamOfSet, ts.IndexOfSet);
133                     return;
134                 }
135                 catch (TargetInvocationException)
136                 {
137                     triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
138                     return;
139                 }
140                 catch (Exception e)
141                 {
142                     TraceWarning(444,
143                         "Schreiben␣des␣statischen␣Feldes␣fehlgeschlagen.", e);
144                     return;
145                 }
146             }
147             triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
148         }
149
150         /// <summary>
151         /// Gets the interaction symbol1.
152         /// </summary>
153         /// <returns></returns>
154         protected override Image GetInteractionSymbol1()
155         {
156             return InteractionSymbols.Images["Field_Static"];
157         }
158
159         /// <summary>
160         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
161         /// welches das gekapselte .NET-Element repräsentieren.
162         /// </summary>
163         /// <returns>
164         /// Das zweite von zwei Symbolen oder <c>null</c>.
165         /// </returns>
166         protected override Image GetInteractionSymbol2()
167         {
168             return InteractionSymbols.Images["Push"];
169         }
170     }
171 }
```

Listing A.21: DynamicNode.Lib.Objects.AbstractProperty

```
#region file header

// <copyright>
// Copyright (c) 2009-2010 by Tobias Kiertscher
// Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
// </copyright>
// <license>DNLicense.html</license>
// <revision>$Revision: 1476 $</revision>
// <author>$Author: Kiertscher.Tobias $</author>
// <date>$Date: 2010-04-26 15:56:29 +0200 (Mo, 26. Apr 2010) $</date>

#endregion

using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using DynamicNode.Core;
using DynamicNode.Ext.Objects;
using DynamicNode.Ext.Visual;
using DynamicNode.Persistence;

namespace DynamicNode.Lib.Objects
{
    public abstract class AbstractProperty : AbstractWrapper
    {
        protected static readonly object[] EMPTY_OBJECT_ARRAY = new object[] { };

        private static PropertySelector selector;
        protected InPort triggerIP;
        protected OutPort triggerOP;

        protected PropertyInfo Property { get; private set; }

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="AbstractProperty"/>.
        /// </summary>
        protected AbstractProperty()
        {
        }

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="AbstractProperty"/>.
        /// </summary>
        /// <param name="propertyInfo">Ein zu kapselndes CLR-Property.</param>
        protected AbstractProperty(PropertyInfo propertyInfo)
        {
            CheckPropertyInfo(propertyInfo);
            Property = propertyInfo;
            InitPorts();
        }

        /// <summary>
        /// Initialisiert die aktuelle Instanz.
        /// Wird von den Konstruktoren aufgerufen.
        /// </summary>
        protected override void Init()
        {
            base.Init();
            triggerIP = new InPort(this, "triggerIn");
            triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
        }

        /// <summary>
        /// Überprüft ob das übergebene Property geeignet ist.
        /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
        /// geworfen.
        /// </summary>
        /// <param name="pI">Das zu prüfende Property.</param>
        protected abstract void CheckPropertyInfo(PropertyInfo pI);

        /// <summary>
        /// Initialisiert den <see cref="PropertySelector"/>, der als
        /// Knotensteuerung dient.
        /// </summary>
        /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
        protected abstract void InitSelector(PropertySelector selector);

        /// <summary>
        /// Initialisiert die Knoten-Steuerung.
        /// </summary>
        /// <param name="ctrlPanelExt">Die Knoten-Steuerung.</param>
        protected override void InitControlPanel(NodeControlPanelExtension ctrlPanelExt)
        {
            if (selector == null)
            {
```

```
87                  selector = new PropertySelector();
88                  selector.Dock = DockStyle.Fill;
89              }
90              ctrlPanelExt.Control = selector;
91              ctrlPanelExt.Show += ctrlPanelExt_Show;
92              ctrlPanelExt.Hide += ctrlPanelExt_Hide;
93          }
94
95          private void ctrlPanelExt_Show(object sender, EventArgs e)
96          {
97              InitSelector(selector);
98              selector.Property = Property;
99              selector.PropertySelected += Selector_PropertySelected;
100         }
101
102         private void ctrlPanelExt_Hide(object sender, EventArgs e)
103         {
104             selector.PropertySelected -= Selector_PropertySelected;
105         }
106
107         private void Selector_PropertySelected(object sender, EventArgs e)
108         {
109             var p = ((PropertySelector)sender).Property;
110             if (p != null)
111             {
112                 try
113                 {
114                     CheckPropertyInfo(p);
115                 }
116                 catch (Exception exc)
117                 {
118                     TraceWarning(430, "Ungültiges␣Property␣ausgewählt.", exc);
119                     p = null;
120                 }
121             }
122
123             Property = p;
124
125             InitPorts();
126             UpdateVisualization();
127         }
128
129         /// <summary>
130         /// Wird aufgerufen wenn der Graph gespeichert wird.
131         /// Wer diese Methode überschreibt muss <c>base.OnStore(pr)</c> aufrufen.
132         /// </summary>
133         /// <param name="pw">Der <see cref="IPropertyWriter"/> mit dem benutzerdefinierte
134         /// Eigenschaften des Knotens gespeichert werden können.</param>
135         public override void OnStore(IPropertyWriter pw)
136         {
137             base.OnStore(pw);
138             if (Property != null)
139             {
140                 pw.WriteProperty("property",
141                     "\n" + ClrTools.GetDescriptor(Property));
142             }
143         }
144
145         /// <summary>
146         /// Wird aufgerufen wenn der Graph rekonstruiert wird.
147         /// Wer diese Methode überschreibt muss <c>base.OnRestore(pr)</c> aufrufen.
148         /// In dieser Methode sollten die Eingänge wenn möglich mit Standard-Werten
149         /// belegt werden.
150         /// </summary>
151         /// <param name="pr">Der <see cref="IPropertyReader"/> mit dem benutzerdefinierte
152         /// Eigenschaften des Knotens gelesen werden können.</param>
153         /// <remarks>
154         /// Diese Methode wird beim Laden eines Graphen aufgerufen bevor die Verbindungen
155         /// zwischen den Knoten wiederhergestellt werden.
156         /// </remarks>
157         public override void OnRestore(IPropertyReader pr)
158         {
159             base.OnRestore(pr);
160             string descriptor = pr.ReadPropertyAsString("property", null);
161             if (descriptor != null)
162             {
163                 try
164                 {
165                     Property = ClrTools.FindProperty(descriptor);
166                     InitPorts();
167                     if (selector != null)
168                     {
169                         selector.Property = Property;
170                     }
171                 }
172                 catch (Exception e)
173                 {
174                     TraceWarning(431, "Ein␣Property␣wurde␣nicht␣gefunden.", e);
```

```
                }
            }
        }

        /// <summary>
        /// Gibt die erste von drei möglichen Textzeilen zurück,
        /// welche das gekapselte .NET-Element beschreiben.
        /// </summary>
        /// <returns>Die erste von drei Textzeilen.</returns>
        protected override string GetTextLine1()
        {
            return Property != null
                ? ClrTools.GetNamespaceCaption(Property.ReflectedType.Namespace)
                : "";
        }

        /// <summary>
        /// Gibt die zweite von drei möglichen Textzeilen zurück,
        /// welche das gekapselte .NET-Element beschreiben.
        /// </summary>
        /// <returns>Die zweite von drei Textzeilen.</returns>
        protected override string GetTextLine2()
        {
            return Property != null
                ? ClrTools.GetTypeCaption(Property.ReflectedType)
                    + "." + ClrTools.GetPropertyCaption(Property, true)
                : "";
        }

        /// <summary>
        /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
        /// welches das gekapselte .NET-Element repräsentieren.
        /// </summary>
        /// <returns>Das erste von zwei Symbolen.</returns>
        protected override Image GetInteractionSymbol1()
        {
            return InteractionSymbols.Images["Properties"];
        }
    }
}
```

Listing A.22: DynamicNode.Lib.Objects.NGet

```
#region file header

// <copyright>
// Copyright (c) 2009-2010 by Tobias Kiertscher
// Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
// </copyright>
// <license>DNLicense.html</license>
// <revision>$Revision: 1537 $</revision>
// <author>$Author: Kiertscher.Tobias $</author>
// <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>

#endregion

using System;
using System.ComponentModel;
using System.Drawing;
using System.Reflection;
using DynamicNode.Core;
using DynamicNode.Ext.Objects;

namespace DynamicNode.Lib.Objects
{
    [ResourceNodeInfo("CLR")]
    public class NGet : AbstractProperty
    {
        public static NGet Creator(MemberInfo memberInfo)
        {
            return new NGet((PropertyInfo)memberInfo);
        }

        private InPortSingleDataTypeProvider instanceTypeProvider;
        private InPort instanceIP;
        private OutPortDataTypeProvider valueTypeProvider;
        private OutPort valueOP;

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="NGet"/>.
        /// </summary>
        public NGet() { }

        /// <summary>
        /// Initialisiert eine neue Instanz von <see cref="NGet"/> mit
        /// einem vorgegebenen CLR-Property.
```

```
44          /// </summary>
45          /// <param name="propertyInfo">Ein CLR-Property das gekapselt werden soll.</param>
46          public NGet(PropertyInfo propertyInfo) : base(propertyInfo) { }
47
48          /// <summary>
49          /// Initialisiert die aktuelle Instanz.
50          /// Wird von den Konstruktoren aufgerufen.
51          /// </summary>
52          protected override void Init()
53          {
54              base.Init();
55              AttachExtension(instanceTypeProvider = new InPortSingleDataTypeProvider());
56              instanceIP = new InPort(this, "instance", instanceTypeProvider);
57              AttachExtension(valueTypeProvider = new OutPortDataTypeProvider());
58              valueOP = new OutPort(this, "value", valueTypeProvider);
59          }
60
61          /// <summary>
62          /// Überprüft ob das übergebene Property geeignet ist.
63          /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
64          /// geworfen.
65          /// </summary>
66          /// <param name="pI">Das zu prüfende Property.</param>
67          protected override void CheckPropertyInfo(PropertyInfo pI)
68          {
69              if (pI == null)
70              {
71                  throw new ArgumentNullException("pI");
72              }
73              string msg;
74              if (!ClrTools.CheckPropertyGet(pI, false, out msg))
75              {
76                  throw new ArgumentException(msg, "pI");
77              }
78          }
79
80          /// <summary>
81          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
82          /// </summary>
83          protected override void InitPorts()
84          {
85              if (Property != null)
86              {
87                  instanceTypeProvider.SupportedDataType = Property.ReflectedType;
88                  valueTypeProvider.SupportedDataType = Property.PropertyType;
89
90                  ClrTools.ConfigNodesParallelisation(this, Property.ReflectedType);
91              }
92              else
93              {
94                  instanceTypeProvider.SupportedDataType = null;
95                  valueTypeProvider.SupportedDataType = null;
96              }
97
98              UpdateVisualization();
99              //Activate();
100         }
101
102         /// <summary>
103         /// Initialisiert den <see cref="PropertySelector"/>, der als
104         /// Knotensteuerung dient.
105         /// </summary>
106         /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
107         protected override void InitSelector(PropertySelector selector)
108         {
109             selector.MustBeStatic = false;
110             selector.Writeable = false;
111         }
112
113         /// <summary>
114         /// Führt die Knoten-Operation durch.
115         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
116         /// </summary>
117         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
118         /// Tokens aller Eingänge.</param>
119         public override void Work(TokenSet ts)
120         {
121             if (Property == null)
122             {
123                 return;
124             }
125             if (triggerIP.IsConnected
126                 && (triggerIP.CurrentTokenState == TokenState.NotSet
127                     || !triggerIP.IsCurrentTokenNew))
128             {
129                 return;
130             }
131
```

```
132             ts.ExcludeFromMinMaxCalculation(triggerIP.Name);
133             var minTS = ts.MinTokenState;
134 
135             if (minTS != TokenState.NotSet && ts[instanceIP].Value != null)
136             {
137                 var instance = ts[instanceIP].Value;
138                 try
139                 {
140                     var value = Property.GetValue(instance, EMPTY_OBJECT_ARRAY);
141 
142                     valueOP.UpdateToken(value, minTS, ts.StreamOfSet, ts.IndexOfSet);
143                     triggerOP.UpdateToken(
144                         ControlTrigger.Instance,
145                         minTS, ts.StreamOfSet, ts.IndexOfSet);
146                     return;
147                 }
148                 catch (TargetInvocationException)
149                 {
150                     valueOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
151                     return;
152                 }
153                 catch (Exception e)
154                 {
155                     TraceWarning(438, "Lesen␣der␣Eigenschaft␣fehlgeschlagen.", e);
156                     return;
157                 }
158             }
159             valueOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
160             triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
161         }
162 
163         /// <summary>
164         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
165         /// welches das gekapselte .NET-Element repräsentieren.
166         /// </summary>
167         /// <returns>Das erste von zwei Symbolen.</returns>
168         protected override Image GetInteractionSymbol1()
169         {
170             return InteractionSymbols.Images[
171                 Property != null && !Property.CanWrite
172                     ? "Properties_Readonly"
173                     : "Properties"];
174         }
175 
176         /// <summary>
177         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
178         /// welches das gekapselte .NET-Element repräsentieren.
179         /// </summary>
180         /// <returns>
181         /// Das zweite von zwei Symbolen oder <c>null</c>.
182         /// </returns>
183         protected override Image GetInteractionSymbol2()
184         {
185             return Property == null || !Property.CanWrite
186                        ? null
187                        : InteractionSymbols.Images["Pull"];
188         }
189     }
190 }
```

Listing A.23: DynamicNode.Lib.Objects.NSet

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1537 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.ComponentModel;
16 using System.Drawing;
17 using System.Reflection;
18 using DynamicNode.Core;
19 using DynamicNode.Ext.Objects;
20 
21 namespace DynamicNode.Lib.Objects
22 {
23     [ResourceNodeInfo("CLR")]
24     public class NSet : AbstractProperty
```

```
25      {
26          public static NSet Creator(MemberInfo memberInfo)
27          {
28              return new NSet((PropertyInfo)memberInfo);
29          }
30
31          private InPortSingleDataTypeProvider instanceTypeProvider;
32          private InPort instanceIP;
33          private InPortSingleDataTypeProvider valueTypeProvider;
34          private InPort valueIP;
35
36          /// <summary>
37          /// Initialisiert eine neue Instanz von <see cref="NSet"/>.
38          /// </summary>
39          public NSet() { }
40
41          /// <summary>
42          /// Initialisiert eine neue Instanz von <see cref="NSet"/> mit
43          /// einem vorgegebenen CLR-Property.
44          /// </summary>
45          /// <param name="propertyInfo">Ein CLR-Property das gekapselt werden soll.</param>
46          public NSet(PropertyInfo propertyInfo) : base(propertyInfo) { }
47
48          /// <summary>
49          /// Initialisiert die aktuelle Instanz.
50          /// Wird von den Konstruktoren aufgerufen.
51          /// </summary>
52          protected override void Init()
53          {
54              base.Init();
55              AttachExtension(instanceTypeProvider = new InPortSingleDataTypeProvider());
56              instanceIP = new InPort(this, "instance", instanceTypeProvider);
57              AttachExtension(valueTypeProvider = new InPortSingleDataTypeProvider());
58              valueIP = new InPort(this, "value", valueTypeProvider);
59          }
60
61          /// <summary>
62          /// Überprüft ob das übergebene Property geeignet ist.
63          /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
64          /// geworfen.
65          /// </summary>
66          /// <param name="pI">Das zu prüfende Property.</param>
67          protected override void CheckPropertyInfo(PropertyInfo pI)
68          {
69              if (pI == null)
70              {
71                  throw new ArgumentNullException("pI");
72              }
73              string msg;
74              if (!ClrTools.CheckPropertySet(pI, false, out msg))
75              {
76                  throw new ArgumentException(msg, "pI");
77              }
78          }
79
80          /// <summary>
81          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
82          /// </summary>
83          protected override void InitPorts()
84          {
85              if (Property != null)
86              {
87                  instanceTypeProvider.SupportedDataType = Property.ReflectedType;
88                  valueTypeProvider.SupportedDataType = Property.PropertyType;
89
90                  ClrTools.ConfigNodesParallelisation(this, Property.ReflectedType);
91              }
92              else
93              {
94                  instanceTypeProvider.SupportedDataType = null;
95                  valueTypeProvider.SupportedDataType = null;
96              }
97
98              UpdateVisualization();
99              //Activate();
100         }
101
102         /// <summary>
103         /// Initialisiert den <see cref="PropertySelector"/>, der als
104         /// Knotensteuerung dient.
105         /// </summary>
106         /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
107         protected override void InitSelector(PropertySelector selector)
108         {
109             selector.MustBeStatic = false;
110             selector.Writeable = true;
111         }
112
```

```
113         /// <summary>
114         /// Führt die Knoten-Operation durch.
115         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
116         /// </summary>
117         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
118         /// Tokens aller Eingänge.</param>
119         public override void Work(TokenSet ts)
120         {
121             if (Property == null)
122             {
123                 return;
124             }
125             if (triggerIP.IsConnected
126                 && (triggerIP.CurrentTokenState == TokenState.NotSet
127                     || !triggerIP.IsCurrentTokenNew))
128             {
129                 return;
130             }
131
132             ts.ExcludeFromMinMaxCalculation(triggerIP.Name);
133             var minTS = ts.MinTokenState;
134
135             if (minTS > TokenState.NotSet && ts[instanceIP].Value != null)
136             {
137                 var instance = ts[instanceIP].Value;
138                 var value = ts[valueIP].Value;
139                 try
140                 {
141                     Property.SetValue(instance, value, EMPTY_OBJECT_ARRAY);
142
143                     triggerOP.UpdateToken(
144                         ControlTrigger.Instance, minTS,
145                         ts.StreamOfSet, ts.IndexOfSet);
146                     return;
147                 }
148                 catch (TargetInvocationException)
149                 {
150                     return;
151                 }
152                 catch (Exception e)
153                 {
154                     TraceWarning(440, "Schreiben␣der␣Eigenschaft␣fehlgeschlagen.", e);
155                     return;
156                 }
157             }
158             triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
159         }
160
161         /// <summary>
162         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
163         /// welches das gekapselte .NET-Element repräsentieren.
164         /// </summary>
165         /// <returns>Das zweite von zwei Symbolen oder <c>null</c>.</returns>
166         protected override Image GetInteractionSymbol2()
167         {
168             return InteractionSymbols.Images["Push"];
169         }
170     }
171 }
```

Listing A.24: DynamicNode.Lib.Objects.NStaticGet

```
1  #region file header
2
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1529 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-21 13:14:11 +0200 (Fr, 21. Mai 2010) $</date>
11
12 #endregion
13
14 using System;
15 using System.Drawing;
16 using System.Reflection;
17 using DynamicNode.Core;
18 using DynamicNode.Ext.Objects;
19
20 namespace DynamicNode.Lib.Objects
21 {
22     [ResourceNodeInfo("CLR")]
23     public class NStaticGet : AbstractProperty
24     {
```

```
25          public static NStaticGet Creator(MemberInfo memberInfo)
26          {
27              return new NStaticGet((PropertyInfo)memberInfo);
28          }
29
30          private OutPortDataTypeProvider valueTypeProvider;
31          private OutPort valueOP;
32
33          /// <summary>
34          /// Initialisiert eine neue Instanz von <see cref="NStaticGet"/>.
35          /// </summary>
36          public NStaticGet() { }
37
38          /// <summary>
39          /// Initialisiert eine neue Instanz von <see cref="NStaticGet"/> mit
40          /// einem vorgegebenen CLR-Property.
41          /// </summary>
42          /// <param name="propertyInfo">Ein CLR-Property das gekapselt werden soll.</param>
43          public NStaticGet(PropertyInfo propertyInfo) : base(propertyInfo) { }
44
45          /// <summary>
46          /// Initialisiert die aktuelle Instanz.
47          /// Wird von den Konstruktoren aufgerufen.
48          /// </summary>
49          protected override void Init()
50          {
51              base.Init();
52              AttachExtension(valueTypeProvider = new OutPortDataTypeProvider());
53              valueOP = new OutPort(this, "value", valueTypeProvider);
54          }
55
56          /// <summary>
57          /// Überprüft ob das übergebene Property geeignet ist.
58          /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
59          /// geworfen.
60          /// </summary>
61          /// <param name="pI">Das zu prüfende Property.</param>
62          protected override void CheckPropertyInfo(PropertyInfo pI)
63          {
64              if (pI == null)
65              {
66                  throw new ArgumentNullException("pI");
67              }
68              string msg;
69              if (!ClrTools.CheckPropertyGet(pI, true, out msg))
70              {
71                  throw new ArgumentException(msg, "pI");
72              }
73          }
74
75          /// <summary>
76          /// Initialisiert den <see cref="PropertySelector"/>, der als
77          /// Knotensteuerung dient.
78          /// </summary>
79          /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
80          protected override void InitSelector(PropertySelector selector)
81          {
82              selector.MustBeStatic = true;
83              selector.Writeable = false;
84          }
85
86          /// <summary>
87          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
88          /// </summary>
89          protected override void InitPorts()
90          {
91              valueTypeProvider.SupportedDataType =
92                  Property != null
93                      ? Property.PropertyType
94                      : null;
95
96              UpdateVisualization();
97              Activate();
98          }
99
100         /// <summary>
101         /// Führt die Knoten-Operation durch.
102         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
103         /// </summary>
104         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
105         /// Tokens aller Eingänge.</param>
106         public override void Work(TokenSet ts)
107         {
108             if (Property == null)
109             {
110                 return;
111             }
112             if (triggerIP.IsConnected
```

```
113                 && (triggerIP.CurrentTokenState == TokenState.NotSet
114                     || !triggerIP.IsCurrentTokenNew))
115             {
116                 return;
117             }
118 
119             try
120             {
121                 var value = Property.GetValue(null, EMPTY_OBJECT_ARRAY);
122 
123                 valueOP.UpdateToken(value, TokenState.Valid, ts.StreamOfSet, ts.IndexOfSet)
                        ;
124                 triggerOP.UpdateToken(
125                     ControlTrigger.Instance,
126                     TokenState.Valid, ts.StreamOfSet, ts.IndexOfSet);
127                 return;
128             }
129             catch (TargetInvocationException)
130             {
131                 valueOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
132                 return;
133             }
134             catch (Exception e)
135             {
136                 TraceWarning(441,
137                              "Lesen␣der␣statischen␣Eigenschaft␣fehlgeschlagen.", e);
138                 return;
139             }
140         }
141 
142         /// <summary>
143         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
144         /// welches das gekapselte .NET-Element repräsentieren.
145         /// </summary>
146         /// <returns>Das erste von zwei Symbolen.</returns>
147         protected override Image GetInteractionSymbol1()
148         {
149             return InteractionSymbols.Images[
150                 Property != null && !Property.CanWrite
151                     ? "Properties_Readonly_Static"
152                     : "Properties_Static"];
153         }
154 
155         /// <summary>
156         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
157         /// welches das gekapselte .NET-Element repräsentieren.
158         /// </summary>
159         /// <returns> Das zweite von zwei Symbolen oder <c>null</c>. </returns>
160         protected override Image GetInteractionSymbol2()
161         {
162             return Property == null || !Property.CanWrite
163                        ? null
164                        : InteractionSymbols.Images["Pull"];
165         }
166     }
167 }
```

Listing A.25: DynamicNode.Lib.Objects.NStaticSet

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1537 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-24 20:08:07 +0200 (Mo, 24. Mai 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.Drawing;
16 using System.Reflection;
17 using DynamicNode.Core;
18 using DynamicNode.Ext.Objects;
19 
20 namespace DynamicNode.Lib.Objects
21 {
22     [ResourceNodeInfo("CLR")]
23     public class NStaticSet : AbstractProperty
24     {
25         public static NStaticSet Creator(MemberInfo memberInfo)
26         {
27             return new NStaticSet((PropertyInfo) memberInfo);
```

```
28          }
29
30          private InPortSingleDataTypeProvider valueTypeProvider;
31          private InPort valueIP;
32
33          /// <summary>
34          /// Initialisiert eine neue Instanz von <see cref="NStaticSet"/>.
35          /// </summary>
36          public NStaticSet(){}
37
38          /// <summary>
39          /// Initialisiert eine neue Instanz von <see cref="NStaticSet"/> mit
40          /// einem vorgegebenen CLR-Property.
41          /// </summary>
42          /// <param name="propertyInfo">Ein CLR-Property das gekapselt werden soll.</param>
43          public NStaticSet(PropertyInfo propertyInfo): base(propertyInfo){}
44
45          /// <summary>
46          /// Initialisiert die aktuelle Instanz.
47          /// Wird von den Konstruktoren aufgerufen.
48          /// </summary>
49          protected override void Init()
50          {
51              base.Init();
52              AttachExtension(valueTypeProvider = new InPortSingleDataTypeProvider());
53              valueIP = new InPort(this, "value", valueTypeProvider);
54          }
55
56          /// <summary>
57          /// Überprüft ob das übergebene Property geeignet ist.
58          /// Falls das Property ungeeignet ist wird eine <see cref="ArgumentException"/>
59          /// geworfen.
60          /// </summary>
61          /// <param name="pI">Das zu prüfende Property.</param>
62          protected override void CheckPropertyInfo(PropertyInfo pI)
63          {
64              if (pI == null)
65              {
66                  throw new ArgumentNullException("pI");
67              }
68              string msg;
69              if (!ClrTools.CheckPropertySet(pI, true, out msg))
70              {
71                  throw new ArgumentException(msg, "pI");
72              }
73          }
74
75          /// <summary>
76          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
77          /// </summary>
78          protected override void InitPorts()
79          {
80              valueTypeProvider.SupportedDataType =
81                  Property != null
82                      ? Property.PropertyType
83                      : null;
84
85              UpdateVisualization();
86              //Activate();
87          }
88
89          /// <summary>
90          /// Initialisiert den <see cref="PropertySelector"/>, der als
91          /// Knotensteuerung dient.
92          /// </summary>
93          /// <param name="selector">Das Steuerelement zur Auswahl eines Properties.</param>
94          protected override void InitSelector(PropertySelector selector)
95          {
96              selector.MustBeStatic = true;
97              selector.Writeable = true;
98          }
99
100         /// <summary>
101         /// Führt die Knoten-Operation durch.
102         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
103         /// </summary>
104         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
105         /// Tokens aller Eingänge.</param>
106         public override void Work(TokenSet ts)
107         {
108             if (Property == null)
109             {
110                 return;
111             }
112             if (triggerIP.IsConnected
113                 && (triggerIP.CurrentTokenState == TokenState.NotSet
114                     || !triggerIP.IsCurrentTokenNew))
115             {
```

```
116                 return;
117             }
118 
119             var vt = ts[valueIP];
120             if (vt.State != TokenState.NotSet)
121             {
122                 try
123                 {
124                     Property.SetValue(null, vt.Value, EMPTY_OBJECT_ARRAY);
125 
126                     triggerOP.UpdateToken(
127                         ControlTrigger.Instance,
128                         vt.State, ts.StreamOfSet, ts.IndexOfSet);
129                     return;
130                 }
131                 catch (TargetInvocationException)
132                 {
133                     return;
134                 }
135                 catch (Exception e)
136                 {
137                     TraceWarning(443,
138                         "Schreiben␣der␣statischen␣Eigenschaft␣fehlgeschlagen.", e);
139                     return;
140                 }
141             }
142             triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
143         }
144 
145         /// <summary>
146         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
147         /// welches das gekapselte .NET-Element repräsentieren.
148         /// </summary>
149         /// <returns>Das erste von zwei Symbolen.</returns>
150         protected override Image GetInteractionSymbol1()
151         {
152             return InteractionSymbols.Images["Properties_Static"];
153         }
154 
155         /// <summary>
156         /// Gibt ein Bild für das zweite der zwei möglichen Symbole zurück,
157         /// welches das gekapselte .NET-Element repräsentieren.
158         /// </summary>
159         /// <returns>
160         /// Das zweite von zwei Symbolen oder <c>null</c>.
161         /// </returns>
162         protected override Image GetInteractionSymbol2()
163         {
164             return InteractionSymbols.Images["Push"];
165         }
166     }
167 }
```

Listing A.26: DynamicNode.Lib.Objects.AbstractEvent

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1529 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-05-21 13:14:11 +0200 (Fr, 21. Mai 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.Collections.Generic;
16 using System.ComponentModel;
17 using System.Drawing;
18 using System.Reflection;
19 using System.Windows.Forms;
20 using DynamicNode.Core;
21 using DynamicNode.Ext.Objects;
22 using DynamicNode.Ext.Visual;
23 using DynamicNode.Persistence;
24 
25 namespace DynamicNode.Lib.Objects
26 {
27     public abstract class AbstractEvent : AbstractWrapper
28     {
29         private EventInfo eventInfo;
30         private EventHandler handlerDelegate;
31         private readonly Queue<object[]> eventQueue = new Queue<object[]>(2);
```

```
32          private OutPort senderOP;
33          private OutPortDataTypeProvider argsTypeProvider;
34          private OutPort argsOP;
35          private OutPort triggerOP;
36          private object lastInstance;
37          private static EventSelector selector;
38
39          /// <summary>
40          /// Initialisiert eine neue Instanz von <see cref="AbstractEvent"/>.
41          /// </summary>
42          protected AbstractEvent() { }
43
44          /// <summary>
45          /// Initialisiert eine neue Instanz von <see cref="AbstractEvent"/>.
46          /// </summary>
47          /// <param name="eventInfo">The event info.</param>
48          protected AbstractEvent(EventInfo eventInfo)
49          {
50              Event = eventInfo;
51          }
52
53          /// <summary>
54          /// Initialisiert die aktuelle Instanz.
55          /// Wird von den Konstruktoren aufgerufen.
56          /// </summary>
57          protected override void Init()
58          {
59              base.Init();
60              triggerOP = new OutPort(this, "triggerOut", typeof(ControlTrigger));
61              senderOP = new OutPort(this, "sender", typeof(object));
62              argsTypeProvider = new OutPortDataTypeProvider();
63              argsOP = new OutPort(this, "eventArgs", argsTypeProvider);
64              handlerDelegate = Handler;
65          }
66
67          /// <summary>
68          /// Initialisiert den <see cref="EventSelector"/>, der als
69          /// Knotensteuerung dient.
70          /// </summary>
71          /// <param name="selector">Das Steuerelement zur Auswahl eines Events.</param>
72          protected abstract void InitSelector(EventSelector selector);
73
74          /// <summary>
75          /// Gibt das Metadaten-Objekt für das gekapselte Ereignis zurück oder legt es fest.
76          /// </summary>
77          /// <value>Die Ereignis-Metatdaten.</value>
78          protected EventInfo Event
79          {
80              get { return eventInfo; }
81              private set
82              {
83                  if (eventInfo == value) return;
84                  try
85                  {
86                      if (value != null)
87                      {
88                          CheckEventInfo(value);
89                      }
90
91                      var oldEventInfo = eventInfo;
92                      eventInfo = value;
93
94                      if (selector != null)
95                      {
96                          selector.Event = eventInfo;
97                      }
98                      InitPorts();
99                      UpdateEventBinding(lastInstance, oldEventInfo);
100                     //UpdateVisualization();
101                 }
102                 catch (ArgumentException e)
103                 {
104                     TraceWarning(433, "Ungültiges␣Ereignis␣ausgewählt.", e);
105                 }
106             }
107         }
108
109         /// <summary>
110         /// Überprüft ob das übergebene Ereignis geeignet ist.
111         /// Falls das Ereignis ungeeignet ist wird eine <see cref="ArgumentException"/>
112         /// geworfen.
113         /// </summary>
114         /// <param name="eI">Das zu prüfende Ereignis.</param>
115         protected abstract void CheckEventInfo(EventInfo eI);
116
117         /// <summary>
118         /// Gibt die erste von drei möglichen Textzeilen zurück,
119         /// welche das gekapselte .NET-Element beschreiben.
```

```
120         /// </summary>
121         /// <returns>Die erste von drei Textzeilen.</returns>
122         protected override string GetTextLine1()
123         {
124             return Event != null
125                        ? ClrTools.GetNamespaceCaption(Event.ReflectedType.Namespace)
126                        : "";
127         }
128 
129         /// <summary>
130         /// Gibt die zweite von drei möglichen Textzeilen zurück,
131         /// welche das gekapselte .NET-Element beschreiben.
132         /// </summary>
133         /// <returns>Die zweite von drei Textzeilen.</returns>
134         protected override string GetTextLine2()
135         {
136             return Event != null
137                        ? ClrTools.GetTypeCaption(Event.ReflectedType)
138                            + "." + ClrTools.GetEventCaption(Event, true)
139                        : "";
140         }
141 
142         /// <summary>
143         /// Initialisiert die Knoten-Steuerung.
144         /// </summary>
145         /// <param name="ctrlPanelExt">Die Knoten-Steuerung.</param>
146         protected override void InitControlPanel(NodeControlPanelExtension ctrlPanelExt)
147         {
148             if (selector == null)
149             {
150                 selector = new EventSelector();
151                 selector.Dock = DockStyle.Fill;
152             }
153 
154             ctrlPanelExt.Control = selector;
155             ctrlPanelExt.Show += CtrlPanelExtShow;
156             ctrlPanelExt.Hide += CtrlPanelExtHide;
157         }
158 
159         private void CtrlPanelExtShow(object sender, EventArgs e)
160         {
161             InitSelector(selector);
162             selector.Event = Event;
163             selector.EventSelected += EventSelector_EventSelected;
164         }
165 
166         private void CtrlPanelExtHide(object sender, EventArgs e)
167         {
168             selector.EventSelected -= EventSelector_EventSelected;
169         }
170 
171         private void EventSelector_EventSelected(object sender, EventArgs e)
172         {
173             Event = ((EventSelector)sender).Event;
174         }
175 
176         protected void UpdateEventBinding(object instance, EventInfo oldEventInfo)
177         {
178             if (lastInstance == instance && oldEventInfo == eventInfo)
179             {
180                 return;
181             }
182             if (oldEventInfo != null &&
183                 (oldEventInfo.GetRemoveMethod().IsStatic || lastInstance != null))
184             {
185                 oldEventInfo.RemoveEventHandler(lastInstance, handlerDelegate);
186                 TraceVerbose(435, "Event␣Handler␣entfernt.");
187             }
188             if (Event != null &&
189                 (Event.GetAddMethod().IsStatic || instance != null))
190             {
191                 Event.AddEventHandler(instance, handlerDelegate);
192                 TraceVerbose(434, "Event␣Handler␣angefügt.");
193             }
194             lastInstance = instance;
195         }
196 
197         /// <summary>
198         /// Führt die Knoten-Operation durch.
199         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
200         /// </summary>
201         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
202         /// Tokens aller Eingänge.</param>
203         public override void Work(TokenSet ts)
204         {
205             lock (eventQueue)
206             {
207                 if (eventQueue.Count > 0)
```

```
208                 {
209                     var args = eventQueue.Dequeue();
210 
211                     senderOP.UpdateToken(args[0]);
212                     argsOP.UpdateToken(args[1]);
213 
214                     triggerOP.UpdateToken(ControlTrigger.Instance,
215                         ts.StreamOfSet, ts.IndexOfSet);
216 
217                     if (eventQueue.Count > 0)
218                     {
219                         Reactivate = true;
220                     }
221                 }
222                 else
223                 {
224                     senderOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
225                     argsOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
226                     //triggerOP.UpdateTokenNotSet(ts.StreamOfSet, ts.IndexOfSet);
227                 }
228             }
229         }
230 
231         /// <summary>
232         /// Der Ereignis-Handler.
233         /// </summary>
234         /// <param name="sender">Der Sender des Ereignisses.</param>
235         /// <param name="e">Eine Instanz von <see cref="System.EventArgs"/>, welche die
                Ereignisparameter enthält.</param>
236         private void Handler(object sender, EventArgs e)
237         {
238             TraceVerbose(436, "Ereignis␣empfangen.");
239             lock (eventQueue)
240             {
241                 eventQueue.Enqueue(new[] { sender, e });
242             }
243             Activate();
244         }
245 
246         /// <summary>
247         /// Wird aufgerufen wenn der Graph gespeichert wird.
248         /// Wer diese Methode überschreibt muss <c>base.OnStore(pr)</c> aufrufen.
249         /// </summary>
250         /// <param name="pw">Der <see cref="IPropertyWriter"/> mit dem benutzerdefinierte
251         /// Eigenschaften des Knotens gespeichert werden können.</param>
252         public override void OnStore(IPropertyWriter pw)
253         {
254             base.OnStore(pw);
255             if (Event != null)
256             {
257                 pw.WriteProperty("event",
258                     "\n" + ClrTools.GetDescriptor(Event));
259             }
260         }
261 
262         /// <summary>
263         /// Wird aufgerufen wenn der Graph rekonstruiert wird.
264         /// Wer diese Methode überschreibt muss <c>base.OnRestore(pr)</c> aufrufen.
265         /// In dieser Methode sollten die Eingänge wenn möglich mit Standard-Werten
266         /// belegt werden.
267         /// </summary>
268         /// <param name="pr">Der <see cref="IPropertyReader"/> mit dem benutzerdefinierte
269         /// Eigenschaften des Knotens gelesen werden können.</param>
270         /// <remarks>
271         /// Diese Methode wird beim Laden eines Graphen aufgerufen bevor die Verbindungen
272         /// zwischen den Knoten wiederhergstellt werden.
273         /// </remarks>
274         public override void OnRestore(IPropertyReader pr)
275         {
276             base.OnRestore(pr);
277             string descriptor = pr.ReadPropertyAsString("event", null);
278             if (descriptor != null)
279             {
280                 try
281                 {
282                     Event = ClrTools.FindEvent(descriptor);
283                 }
284                 catch (Exception e)
285                 {
286                     TraceWarning(437, "Ein␣Event␣wurde␣nicht␣gefunden.", e);
287                 }
288             }
289         }
290 
291         /// <summary>
292         /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
293         /// </summary>
294         protected override void InitPorts()
```

```
295         {
296             if (Event != null && Event.EventHandlerType != null)
297             {
298                 var mi = Event.EventHandlerType.GetMethod("Invoke");
299                 var parameters = mi.GetParameters();
300                 argsTypeProvider.SupportedDataType
301                     = parameters.Length != 2
302                           ? null
303                           : parameters[1].ParameterType;
304 
305                 ClrTools.ConfigNodesParallelisation(this, Event.ReflectedType);
306             }
307             else
308             {
309                 argsTypeProvider.SupportedDataType = null;
310             }
311             UpdateVisualization();
312         }
313 
314         /// <summary>
315         /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
316         /// welches das gekapselte .NET-Element repräsentieren.
317         /// </summary>
318         /// <returns>Das erste von zwei Symbolen.</returns>
319         protected override Image GetInteractionSymbol1()
320         {
321             return InteractionSymbols.Images["Event"];
322         }
323     }
324 }
```

Listing A.27: DynamicNode.Lib.Objects.NEvent

```
1  #region file header
2  
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1476 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-04-26 15:56:29 +0200 (Mo, 26. Apr 2010) $</date>
11 
12 #endregion
13 
14 using System;
15 using System.Reflection;
16 using DynamicNode.Core;
17 using DynamicNode.Ext.Objects;
18 
19 namespace DynamicNode.Lib.Objects
20 {
21     [ResourceNodeInfo("CLR")]
22     public class NEvent : AbstractEvent
23     {
24         public static NEvent Creator(MemberInfo memberInfo)
25         {
26             return new NEvent((EventInfo)memberInfo);
27         }
28 
29         private InPortSingleDataTypeProvider instanceTypeProvider;
30         private InPort instanceIP;
31 
32         /// <summary>
33         /// Initialisiert eine neue Instanz von <see cref="NEvent"/>.
34         /// </summary>
35         public NEvent() { }
36 
37         /// <summary>
38         /// Initialisiert eine neue Instanz von <see cref="NEvent"/>.
39         /// </summary>
40         /// <param name="eventInfo">Ein CLR-Event das gekapselt werden soll.</param>
41         public NEvent(EventInfo eventInfo) : base(eventInfo) { }
42 
43         /// <summary>
44         /// Überprüft ob das übergebene Ereignis geeignet ist.
45         /// Falls das Ereignis ungeeignet ist wird eine <see cref="ArgumentException"/>
46         /// geworfen.
47         /// </summary>
48         /// <param name="eI">Das zu prüfende Ereignis.</param>
49         protected override void CheckEventInfo(EventInfo eI)
50         {
51             if (eI == null)
52             {
53                 throw new ArgumentNullException("eI");
```

```
54              }
55              if (eI.GetAddMethod().IsStatic || eI.GetRemoveMethod().IsStatic)
56              {
57                  throw new ArgumentException("Das␣zu␣kapselnde␣Ereignis␣ist␣statisch.", "eI"
                        );
58              }
59          }
60
61          /// <summary>
62          /// Initialisiert die aktuelle Instanz.
63          /// Wird von den Konstruktoren aufgerufen.
64          /// </summary>
65          protected override void Init()
66          {
67              base.Init();
68              instanceTypeProvider = new InPortSingleDataTypeProvider();
69              instanceIP = new InPort(this, "instance", instanceTypeProvider);
70          }
71
72          /// <summary>
73          /// Initialisiert den <see cref="EventSelector"/>, der als
74          /// Knotensteuerung dient.
75          /// </summary>
76          /// <param name="selector">Das Steuerelement zur Auswahl eines Events.</param>
77          protected override void InitSelector(EventSelector selector)
78          {
79              selector.ShowStaticEvents = false;
80          }
81
82          /// <summary>
83          /// Initialisiert oder aktualisiert die Anschlüsse nach Auswahl eines Properties.
84          /// </summary>
85          protected override void InitPorts()
86          {
87              if (Event != null && Event.EventHandlerType != null)
88              {
89                  instanceTypeProvider.SupportedDataType = Event.ReflectedType;
90              }
91              else
92              {
93                  instanceTypeProvider.SupportedDataType = null;
94              }
95              base.InitPorts();
96          }
97
98          /// <summary>
99          /// Führt die Knoten-Operation durch.
100         /// Dabei können die Ausgänge mit neuen Tokens belegt werden.
101         /// </summary>
102         /// <param name="ts">Das <see cref="TokenSet"/> mit den aktuellen
103         /// Tokens aller Eingänge.</param>
104         public override void Work(TokenSet ts)
105         {
106             object instance = null;
107
108             var t = ts[instanceIP];
109             if (t.State > TokenState.NotSet)
110             {
111                 instance = t.Value;
112             }
113             UpdateEventBinding(instance, Event);
114
115             base.Work(ts);
116         }
117     }
118 }
```

Listing A.28: DynamicNode.Lib.Objects.NStaticEvent

```
1  #region file header
2
3  // <copyright>
4  // Copyright (c) 2009-2010 by Tobias Kiertscher
5  // Alle Rechte vorbehalten, soweit in Lizenz nicht anders bestimmt.
6  // </copyright>
7  // <license>DNLicense.html</license>
8  // <revision>$Revision: 1465 $</revision>
9  // <author>$Author: Kiertscher.Tobias $</author>
10 // <date>$Date: 2010-04-21 13:39:39 +0200 (Mi, 21. Apr 2010) $</date>
11
12 #endregion
13
14 using System;
15 using System.Drawing;
16 using System.Reflection;
17 using DynamicNode.Core;
```

```
18  using DynamicNode.Ext.Objects;
19
20  namespace DynamicNode.Lib.Objects
21  {
22      [ResourceNodeInfo("CLR")]
23      public class NStaticEvent : AbstractEvent
24      {
25          public static NStaticEvent Creator(MemberInfo memberInfo)
26          {
27              return new NStaticEvent((EventInfo)memberInfo);
28          }
29
30          /// <summary>
31          /// Initialisiert eine neue Instanz von <see cref="NEvent"/>.
32          /// </summary>
33          public NStaticEvent() { }
34
35          /// <summary>
36          /// Initialisiert eine neue Instanz von <see cref="NEvent"/>.
37          /// </summary>
38          /// <param name="eventInfo">Ein CLR-Event das gekapselt werden soll.</param>
39          public NStaticEvent(EventInfo eventInfo) : base(eventInfo) { }
40
41          /// <summary>
42          /// Überprüft ob das übergebene Ereignis geeignet ist.
43          /// Falls das Ereignis ungeeignet ist wird eine <see cref="ArgumentException"/>
44          /// geworfen.
45          /// </summary>
46          /// <param name="eI">Das zu prüfende Ereignis.</param>
47          protected override void CheckEventInfo(EventInfo eI)
48          {
49              if (eI == null)
50              {
51                  throw new ArgumentNullException("eI");
52              }
53              if (!eI.GetAddMethod().IsStatic || !eI.GetRemoveMethod().IsStatic)
54              {
55                  throw new ArgumentException("Das␣zu␣kapselnde␣Ereignis␣ist␣nicht␣statisch."
                        , "eI");
56              }
57          }
58
59          /// <summary>
60          /// Initialisiert den <see cref="EventSelector"/>, der als
61          /// Knotensteuerung dient.
62          /// </summary>
63          /// <param name="selector">Das Steuerelement zur Auswahl eines Events.</param>
64          protected override void InitSelector(EventSelector selector)
65          {
66              selector.ShowStaticEvents = true;
67          }
68
69          /// <summary>
70          /// Gibt ein Bild für das erste der zwei möglichen Symbole zurück,
71          /// welches das gekapselte .NET-Element repräsentieren.
72          /// </summary>
73          /// <returns>Das erste von zwei Symbolen.</returns>
74          protected override Image GetInteractionSymbol1()
75          {
76              return InteractionSymbols.Images["Event_Static"];
77          }
78      }
79  }
```

Index

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Das Konzept der Brückenknoten
Einbindung einer imperativen Klassenbibliothek in ein Datenflussprogrammiersystem

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Tobias Kiertscher

Brandenburg/Havel, den 28.05.2010