Diplomarbeit

DynamicNodes

Entwurf und Implementierung eines
Flussgraphen-basierten visuellen Programmiersystems
zur parallelen Ausführung auf Mehrprozessorsystemen

Autor	**Tobias Kiertscher**
Zu erlangender Grad	**Diplom-Informatiker (FH)**
Hochschule	**Fachhochschule Brandenburg**
Fachbereich	**Informatik und Medien**
Ort	**Brandenburg an der Havel / Deutschland**
Erstbetreuer	**Prof. Dr. rer. nat. Friedhelm Mündemann**
Zweitbetreuer	**Prof. Dr. rer. nat. Reiner Creutzburg**
Eingereicht am	**14.08.2007**

Danksagung

Zu allererst möchte ich Gott meinem Schöpfer danken. Er hat mich gemacht und mir die Fähigkeit zum Denken gegeben. Ihm verdanke ich auch mein gutes Verständnis für Technik und Informatik.

„Alle Weisheit fängt damit an, dass wir ihn ernst nehmen. Wer sein Leben nach Gottes Geboten ausrichtet, der allein gewinnt Einsicht.“ Ps. 111,10a

Ich möchte mich auch bei meiner geliebten Frau Susann bedanken. Sie hat mir nicht nur in den Monaten der Arbeit den Rücken gestärkt, mich liebevoll umsorgt und so manches technisches Gerümpel in unserem Wohnzimmer ertragen, sondern hat auch die ganze Arbeit mit bewundernswerter Ausdauer zur Korrektur gelesen, obwohl sich wohl so mancher Satz ihrem Verständnis entzog. ☺

Ohne meinen Tutor Prof. Mündemann wäre diese Arbeit nicht das was sie ist. Er hat mit sanftem Druck das Beste aus mir herausgeholt. Selbst um Mitternacht, stand er mir für Online-Konferenzen zur Verfügung. Herzlichen Dank für Ihre Begeisterung.

Ein herzliches Dankeschön geht auch an meinen Vater Michael, der ebenfalls bei der Korrektur der Arbeit geholfen und mich immer ermutigt hat.

Das letzte Dankeschön geht an meine Mutter Magdalene und meinen Bruder Daniel. Sie haben mir durch ihr unerschütterliches Vertrauen Kraft gegeben.

Inhaltsverzeichnis

Abstract

Diploma thesis from Tobias Kiertscher on the subject: DynamicNodes – draft and implementation of a flow graph based visual programming system for the parallel execution on multi processor systems

Tutors: Prof. Dr. rer. nat. Friedhelm Mündemann and Prof. Dr. rer. nat. Reiner Creutzburg

Keywords: programming system, development environment, runtime environment, flow graph, Microsoft .NET, parallel execution, diploma thesis

Development environments have evolved from simple text editors with compiler support to true feature monsters. Thereby the comfortable production and management for source code was dominating. This thesis wants to strike another path. A visual development environment for production and execution of flow graph based programs with an effective easy to learn development process looks promising.

Based on Microsoft .NET and the programming language C# describes this thesis the draft and the implementation of a development environment with comfortable user interface and efficient runtime environment. It designs a flow model which supports both, the experimental and the productive application of operation nodes of different abstraction levels in nested flow graphs. Operation nodes can work on different data types like literals, values und references. The runtime environment provides for automatically parallel execution on multi-processors.

A help system and a basis for user-friendly functional descriptions of the operation nodes make the entrance for users easier. The programming system can be extended easily by operation nodes in all .NET programming languages.

Zusammenfassung

Diplomarbeit von Tobias Kiertscher zum Thema: DynamicNodes – Entwurf und Implementierung eines Flussgraphen-basierten visuellen Programmiersystems zur parallelen Ausführung auf Mehrprozessorsystemen.

Betreuer: Prof. Dr. rer. nat. Friedhelm Mündemann und Prof. Dr. rer. nat. Reiner Creutzburg

Stichwörter: Programmiersystem, Entwicklungsumgebung, Laufzeitumgebung, Flussgraph, Microsoft .NET, parallele Ausführung, Diplomarbeit

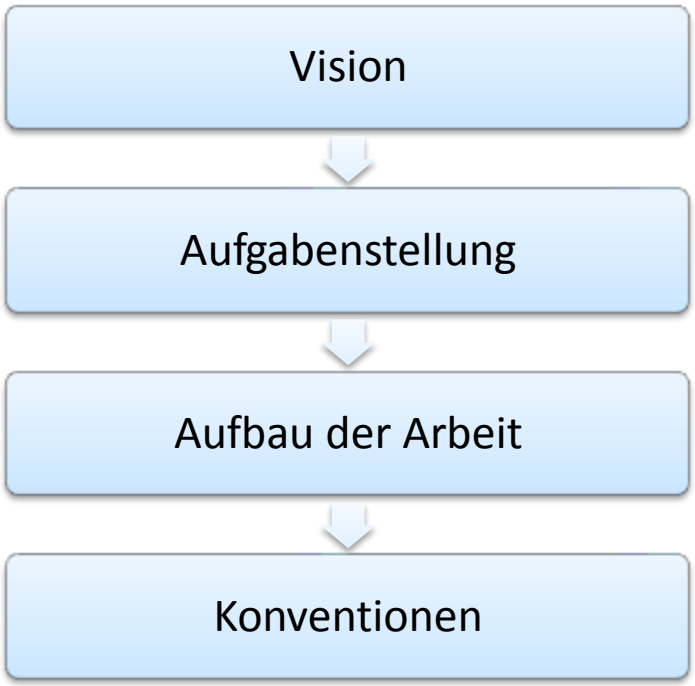
Entwicklungsumgebungen haben sich in den letzten Jahren von einfachen Texteditoren mit Compiler-Unterstützung zu wahren Feature-Monstern entwickelt. Dabei stand die komfortable Erzeugung und Verwaltung von Quelltext im Vordergrund. Diese Arbeit möchte einen anderen Weg einschlagen. Eine visuelle Programmierumgebung zur Erzeugung und Ausführung von Flussgraphen-basierten Programmen verspricht einen leicht zu erlernenden effektiven Entwicklungsprozess.

Auf der Basis von Microsoft .NET und der Programmiersprache C# beschreibt diese Arbeit den Entwurf und die Implementierung einer Programmierumgebung mit komfortabler Benutzeroberfläche und leistungsfähiger Laufzeitumgebung. Sie entwirft ein Flussmodell, welches den experimentellen und produktiven Einsatz von Operationsknoten verschiedener Abstraktionsebenen in verschachtelten Flussgraphen erlaubt. Dabei können Konstanten, Wertdatentypen und Verweistypen verarbeitet werden. Die Programme werden automatisch auf mehreren Prozessoren parallel ausgeführt.

Ein Hilfesystem und die Grundlage für benutzerfreundliche Funktionsbeschreibungen der Operationsknoten erleichtern den Einstieg für den Anwender. Das Programmiersystem kann in allen .NET-Programmiersprachen leicht um Operationsknoten erweitert werden.

1 Einleitung

DynamicNodes – ein visuelles Flussgraphen-basiertes Programmiersystem. Um dem Leser die Thematik ein wenig näher zu bringen, möchte der Autor zu Beginn die Vision teilen, die zu dieser Arbeit geführt hat.
Abschnitt 1.2 enthält die Aufgabenstellung und Abschnitt 1.3 beschreibt den Aufbau der Arbeit.



1.1 Vision

Man stelle sich eine einfache und übersichtliche Benutzeroberfläche für das Programmieren vor. Man vergesse alles was man über Programmiersprachen und Quellcode weiß und denke einen Schritt weiter. Eine umfangreiche aber übersichtliche Anzahl von elementaren Bausteinen verschiedener Abstraktionsebenen steht zur Verfügung, um nahezu jedes Problem mit ein paar Maus-Klicks zu lösen, sofern ein Lösungsweg bekannt ist. Es gibt Funktionseinheiten um Datenquellen anzusprechen, um Bilddaten zu verarbeiten, um Audiodaten zu verarbeiten, um XML-Daten zu verarbeiten, … – und jede Menge Möglichkeiten die verarbeiteten Daten auszugeben oder mit anderen Systemen auszutauschen.

Eine komfortable grafische Schnittstelle bietet eine ausgewogene Mischung aus intelligenter Assistenz und voller Kontrolle über den Entwicklungsprozess. Die erstellten Programme erklären sich von selbst und bleiben in jeder Größenordnung übersichtlich. Knotenpunkte stellen Datenverarbeitungsschritte dar und Verbindungen zwischen ihnen stehen für Datenabhängigkeiten. Eine automatische Skalierung auf Mehrprozessorsystemen ist selbstverständlich. Und jeder Schüler kann ohne Vorkenntnisse, in einer ausgewählten Anwendungsdomäne, Wissen und Erfahrung erlangen und komplizierte Probleme mit Hilfe von eigenen Ideen lösen.

Zugegeben, ein solches System wird es wohl nie geben. Aber diese Arbeit soll versuchen dieser Vision ein Stück näher zu kommen.

1.2 Aufgabenstellung

Ziel der Arbeit ist es, ein Programmiersystem zu entwerfen und zu implementieren, das mit Hilfe einer visuellen Benutzerschnittstelle das Erstellen und Ausführen von Flussgraphen-basierten Programmen ermöglicht. Dazu ist das Flussmodell zu definieren, welches den Programmen zu Grunde liegen soll. Das Programmiersystem muss die erstellten Programme parallel auf mehreren Prozessoren in einem Rechner ausführen können.
Als Ausgangspunkt für den Entwurf des Systems ist eine Spezifikation zu erarbeiten, welche die Entwurfsziele genauer beschreibt. Ein wichtiges Entwurfsziel ist die Benutzbarkeit der Benutzeroberfläche. Und das Programmiersystem sollte erweiterbar sein.
Das Ergebnis der Implementierung ist nach mehreren Kriterien zu bewerten.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 einige theoretische Grundlagen gestreift, welche für den Entwurf des Systems von Bedeutung sind. Dabei werden die Themen *Graphentheorie*, *Architektur von datenverarbeitenden Systemen*, *objektorientierte Programmierung* und das *Microsoft .NET-Framework* aufgegriffen.
Darauf folgt die Spezifikation des Systems in Kapitel 3. In diesem Kapitel werden u.a. das anvisierte Einsatzgebiet des Programmiersystems, funktionale und nicht funktionale Anforderungen aufgeführt.

Der Entwurf des Systems wird in Kapitel 4 dokumentiert. Dabei wird ausgehend von grundlegenden Entwurfsentscheidungen ein Schwerpunkt auf das *Flusskonzept*, das *Datenmodell*, die *Parallelität* und die *Benutzeroberfläche gelegt*.
Kapitel 5 beschreibt das Ergebnis der Implementierung. Die verschiedenen Komponenten des Systems werden vorgestellt und die wichtigsten Klassen in ihrem Kontext erläutert.
Zum Abschluss wird in Kapitel 6 das Ergebnis der Arbeit betrachtet und unter Aspekten wie *Leistungsfähigkeit*, *Typensicherheit* und *Flusssteuerungsmöglichkeiten* betrachtet.
Der Anhang gliedert sich in zwei Teile. Der erste Teil (A-F) ist im Anschluss an Kapitel 6 in diesem Band enthalten. Anhang G enthält die Quelltexte und befindet sich in einem zusätzlichen Band.

1.4 Konventionen

In der Arbeit werden einige Formatierungskonventionen verwendet. Es folgt eine kurze Erläuterung:

- Normaler Text
  Das ist ein ganz normaler Text.
- Schwache Hervorhebung
  Dieser Text soll ohne besondere Hervorhebung von dem normalen Text abgegrenzt werden.
- Hervorhebung
  Dieser Text soll bestimmte Wörter im Text hervorheben. Wichtige Schlüsselwörter werden bei der ersten Verwendung und besonders wichtige Schlüsselwörter immer in dieser Formatierung geschrieben.
- Starke Hervorhebung
  Besonders wichtige Schlüsselwörter werden bei ihrer ersten Verwendung in diesem Format geschrieben.
- Quellcode im Text
  `Mit dieser Formatierung werden Quellcode-Begriffe geschrieben.`
- Quellcode als Absatz

```
Ganze Absätze mit Quellcode werden in dieser Art formatiert.
```

- Beschriftungen
  Alle Abbildungen, Tabellen, Diagramme o.ä. erhalten eine Beschriftung mit dieser Formatierung.
- Überschriften der 4. Ebene

Das ist eine Überschrift auf vierter Ebene

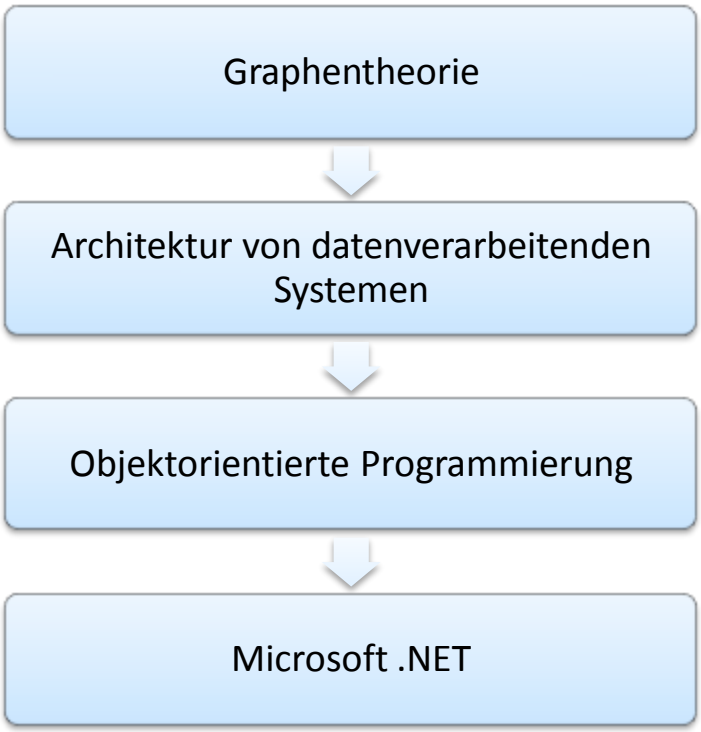
- Überschriften der 5. Ebene

Das ist eine Überschrift auf fünfter Ebene

2 Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für diese Arbeit kurz aufgeführt. Das Kapitel hat nicht zum Ziel, den enthaltenen Stoff lehrbuchartig zu vermitteln, sondern soll lediglich einen Eindruck verschaffen, welche Theorien bei dem Entwurf von DynamicNodes von Bedeutung waren.

Das Kapitel gliedert sich in vier Abschnitte. Der Erste behandelt theoretische Grundlagen der Graphentheorie, der Zweite Theorien der Rechnerarchitektur, der Dritte die objektorientierte Programmierung und der Vierte stellt das .NET-Framework von Microsoft vor.



2.1 Graphentheorie

Die Grundlage für die Graphentheorie gab im Jahre 1736 das Mathematik-Genie Leonard Euler (1707 - 1783) mit einer Veröffentlichung über das sog. Königsberger Brückenproblem. Den heutigen Namen erhielt dieses Teilgebiet der Mathematik jedoch erst 1936, als der ungarische Dénes König (1884 - 1944) das erste Lehrbuch über die „Graphentheorie“ veröffentlichte. In diesem Werk wurden erstmals die vielen Einzelresultate der vorrangegangenen Forschung zusammengetragen. Die Graphentheorie ist heute ein weites Gebiet, welches sowohl in der reinen als auch in der angewandten Mathematik zu Hause ist.

In der vorliegenden Arbeit ist keine ausführliche Darstellung der Graphentheorie enthalten. Der Leser sei dazu auf ein aktuelles Lehrbuch verwiesen, z.B. auf (Volkmann, 1996) an dessen Werk sich diese Arbeit anlehnt. Zum besseren Verständnis sollen an dieser Stelle lediglich einige Grundlagen erwähnt werden.

2.1.1 Definition eines Graphen

Ein Graph besitzt zwei disjunkte Mengen E und K. Dabei bezeichnet E die Menge der im Graph enthaltenen Ecken und K die Menge der Kanten des Graphen.

Wir setzen $P_2(E) = \{X | X \subseteq E \text{ mit } 1 \leq |X| \leq 2\}$, wobei $|X|$ die Kardinalzahl von X bedeutet.

Ein Graph besitzt weiterhin eine auf der Kantenmenge K definierte Abbildung

$$g: K \mapsto P_2(E)$$

Der Graph G ist dann definiert durch das Tripel $G = (E, K, g)$.

Für die Betrachtungen der vorliegenden Arbeit sind lediglich endliche Graphen von Bedeutung, bei denen E und K endliche Mengen sind. Die Endpunkte x, y der Kante $k \in K$ sind gegeben durch $g(k) = \{x, y\}$ wobei x und y nicht notwendig verschieden sind. Graphen können grafisch dargestellt werden, indem die Ecken als Punkte und die Kanten als Verbindungslinien gezeichnet werden.

Eine grafische Darstellung des Graphen $G = (E, K, g)$ mit zwei disjunkten Teilmengen

$$E = \{x_1, x_2, x_3, x_4, x_5\} \text{ und } K = \{k_1, k_2, k_3, k_4, k_5, k_6, k_7\}$$

und einer Abbildung der Kanten

$$g(k_1) = g(k_2) = g(k_3) = \{x_2, x_3\}, g(k_4) = \{x_3, x_4\},$$
$$g(k_5) = \{x_3, x_5\}, g(k_6) = \{x_4, x_5\} \text{ und } g(k_7) = \{x_4\}$$

kann wie in Abb. 1 aussehen.

Ein Graph $G = (E, K, g)$ heißt gerichtet, wenn die zu jedem $k \in K$ durch g zugeordnete Menge geordnet ist, also ein Paar (x, y) darstellt. In einer gerichteten Kante $g(k) = (x, y)$ bezeichnet man x als Start- und y als Endpunkt.

Die Kanten eines gerichteten Graphen werden grafisch als Pfeile dargestellt.

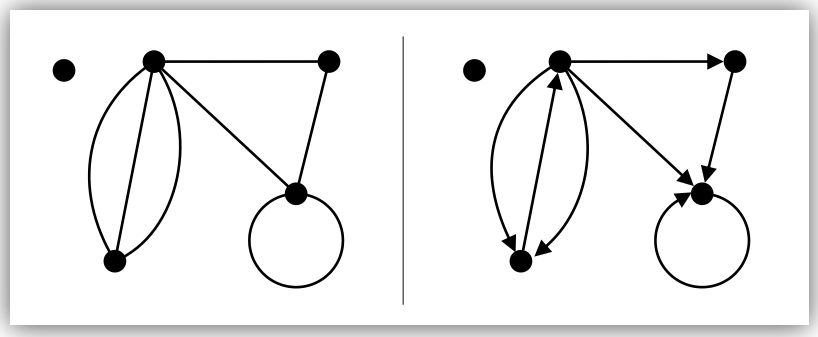


Abb. 1 (v.l.n.r) Ein ungerichteter Graph, ein gerichteter Graph

Homomorphismus und Isomorphismus

Gegeben seien zwei Graphen $G = (E, K, g)$ und $G' = (E', K', g')$ sowie zwei Abbildungen $f: E \to E'$ und $F: K \to K'$. Dann heißt das Paar (f, F) Homomorphismus, wenn für alle $k \in K$ gilt:

$$g(k) = \{x, y\} \Longrightarrow g'\big(F(k)\big) = \{f(x), f(y)\}$$

Für den Graphenhomomorphismus gibt es die Kurzschreibweise $(f, F): G \to G'$.

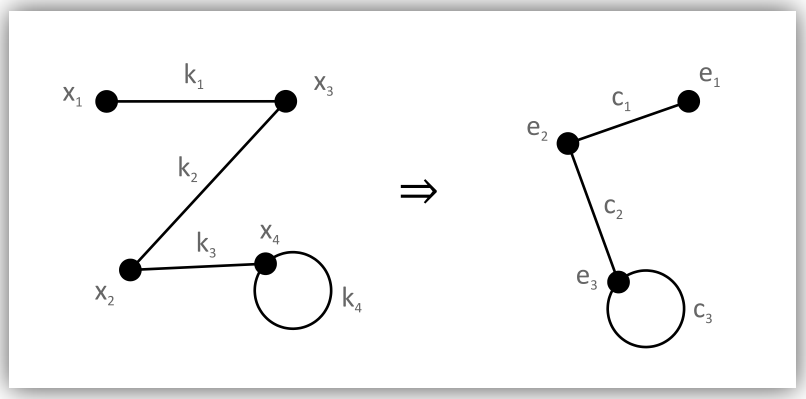


Abb. 2 Graphenhomomorphismus

Sind die Abbildungen f und F bijektiv, so ist (f, F) ein Isomorphismus, bzw. heißen die Graphen G und G' isomorph ($G \cong G'$). Isomorphe Graphen können weitestgehend als gleich angesehen werden.

2.1.2 Flussgraphen

In der aktuellen Literatur wird der Begriff des Flussgraphen mit hauptsächlich zwei unterschiedlichen Bedeutungen gebraucht. Die eine Bedeutung ergibt sich aus der Verwendung eines Graphen zur Analyse von Programmabläufen im imperativen Programmiermodell. Dabei wird der mögliche Kontrollfluss eines sequentiell abzuarbeitenden Programms als gerichteter Graph dargestellt (Ernst, 2000). Ein Flussgraph in diesem Kontext wird auch *Kontrollflussgraph* oder *Flussdiagramm* genannt. Die zweite Bedeutung, in welcher der Begriff Flussgraph verwendet wird, ist die Darstellung der Struktur eines Programms im Programmierparadigma der datenflussorientierten Programmierung (Waldschmidt, et al., 1995). In dieser Arbeit wird der Begriff Flussgraph in seiner zweiten Bedeutung gebraucht, in welcher er auch *Datenflussgraph* genannt wird.

Eine gute Übersicht und Zusammenfassung der Datenflusskonzepte gibt John A. Sharp in (Sharp, 1992). Von seiner Arbeit soll an dieser Stelle profitiert und ebenfalls ein kurzer Überblick gegeben werden.

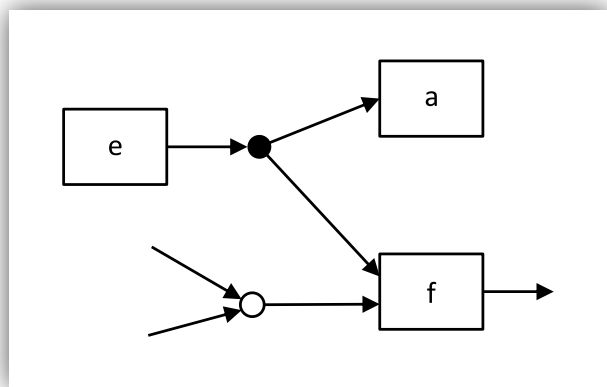


Abb. 3 Skizze eines beispielhaften Flussgraphen

Abb. 3 zeigt ein Beispiel für einen Datenflussgraph. Der Knoten e ist eine Quelle, d.h. er produziert Marken, im Gegensatz dazu ist Knoten a eine Senke die Marken konsumiert aber nicht weitergibt. Knoten f ist ein typischer Funktionsknoten, der Marken auf mehreren eingehenden Verbindungen verarbeitet und Ergebnismarken weitergibt. Der ausgefüllte Kreis ist eine kopierende Verzweigung. Alle Marken, welche auf die eingehende Verbindung gelegt werden, werden kopiert und an alle ausgehenden Verbindungen weitergegeben. Der leere Kreis ist ein zusammenführender Knoten, der alle Marken, die auf eine seiner eingehenden Verbindungen gelegt werden, an die einzige ausgehende Verbindung weitergibt.

Der große Unterschied zwischen dem imperativen und dem datenflussorientierten Programmierparadigma ist die Art und Weise, wie der Programmcode zur Ausführung kommt. Das imperative Programmierparadigma sieht einen sequentiellen Kontrollfluss vor (*control driven*), welcher auf einem Von-Neumann-Rechner Schritt für Schritt nacheinander abgearbeitet wird. Der Zeitpunkt zu dem ein Befehl ausgeführt wird, ist definiert durch die Reihenfolge des Programmcodes. Zu jedem Zeitpunkt kommt immer nur ein Befehl zur Ausführung. Im datenflussorientierten Programmierparadigma werden zwischen Funktionseinheiten Datenabhängigkeiten definiert, welche die Funktionseinheiten miteinander verknüpfen. Wann eine Funktionseinheit aktiv wird, hängt nur davon ab, ob die Datenabhängigkeiten der Funktionseinheit erfüllt sind. In diesem Modell können mehrere Funktionen gleichzeitig abgearbeitet werden.

Die heutige Darstellung von Datenflussgraphen gehen auf die Arbeiten (Adams, 1970), (Dennis, 1974), (Rumbaugh, 1977), (Kosinski, 1973) und einige andere zurück. Es wurden verschiedene Notationen entwickelt, mit jeweils unterschiedlichen elementaren Knoten zur Realisierung von Programmen nach dem Datenfluss-Modell. Die Notationen unterscheiden sich in Punkten wie der Vorbedingung für die Ausführung eines Funktionsknotens oder die mögliche Anzahl von Marken auf einer Kante. Es werden verschiedene Arten von zusätzlichen Kontrollflüssen definiert, um Programmstrukturen wie Schleifen zu ermöglichen.

Man kann Datenflusssysteme nach verschiedenen Kriterien unterscheiden. Es gibt Systeme, welche *data driven* arbeiten und Systeme, welche *demand driven* arbeiten. Bei der ersten Sorte wird ein Funktionsknoten aktiv sobald an allen Eingängen Daten vorhanden sind. Es pflanzt sich sozusagen eine Aktivitätswelle von den Quellen bis zu den Senken des Graphen fort. Bei der zweiten Sorte wer-

den die Funktionsknoten durch eine Nachfrage an ihrem Ausgang aktiv. Diese Nachfragewelle pflanzt sich ebenfalls im Graph fort, jedoch läuft sie umgekehrt, von den Senken zu den Quellen. Der Vorteil dieses Verfahrens liegt darin, dass nur Funktionseinheiten aktiv werden, deren Ergebnisse tatsächlich benötigt werden.

Ein weiteres Kriterium nach dem Datenflusssysteme unterschieden werden können, sind die Art und Weise wie Knoten erzeugt werden. Bei *statischen* Flussgraphen werden die Knoten bei Aufbau des Graphen lediglich einmal erzeugt. Jeder Knoten im Graph kann zu jedem Zeitpunkt nur höchstens einfach aktiv sein. Der Nachteil dieses Prinzips ist, dass Schleifen und Rekursionen schon bei Erzeugung des Graphen dekomponiert werden müssen, um iterativ abgearbeitet werden zu können. Bei *dynamischen* Flussgraphen kann ein Knoten durch mehrere Marken auf seinen Eingangskanten mehrfach aktiv werden. Dazu wird der Knoten zur Laufzeit mehrfach instanziiert.

Damit der nachfolgende Datenfluss mit mehreren Ergebnissen eines Funktionsknotens korrekt ablaufen kann, wird ein zusätzlicher Mechanismus erforderlich. Auch für diesen Mechanismus werden durch die oben genannten Werke unterschiedliche Lösungsansätze vorgestellt. Eine Variante ist das dynamisch-rekursive Verfahren, bei dem für jede neu erzeugte Instanz eines Knotens auch der nachfolgende Sub-Graph kopiert wird. Die zweite Variante wird als das Verfahren der gefärbten Marken bezeichnet und verwendet eine „Färbung“ der Marken auf den Kanten zur Unterscheidung ihrer Identität. Auf den Kanten darf eine große Menge unterschiedlich gefärbter Marken liegen. Wenn an den Eingängen eines Funktionsknotens ausreichend Marken mit der gleichen Färbung vorhanden sind, wird der Knoten aktiv und verarbeitet die entsprechenden Marken. Dieses Verfahren hat den Vorteil, dass kein aufwändiges Kopieren der Sub-Graphen notwendig ist.

Datenflusssysteme unterscheiden sich auch an der Vorbedingung für die Aktivierung eines Funktionsknotens. Einige legen die Voraussetzung zu Grunde, dass ein Knoten nur genau dann aktiviert wird, wenn auf jeder eingehenden Kante mindestens eine Marke liegt, und dass der Knoten von jeder eingehenden Kante nur genau eine Marke konsumiert. Andere erlauben das Aktivieren von Knoten, ohne dass auf jeder eingehenden Kante eine Marke liegt.

Einige Systeme erlauben auch, dass Knoten bei ihrer Aktivierung mehrere Marken von einer eingehenden Kante konsumieren dürfen, was die Grundlage für sog. Streams darstellt.

2.1.3 Aggregation von Graphen zu Knoten

An dieser Stelle soll ein Verfahren vorgestellt werden, welches das verschachtelte Modellieren von Flussgraphen ermöglicht. Das Ziel des Verfahrens ist es, einen Flussgraphen in seiner Gesamtheit als Funktionsknoten zu betrachten und als solchen in weitere Flussgraphen einzubetten. Der Funktionsknoten, dessen Funktion die Ausführung eines gekapselten Graphen ist, wird Aggregationsknoten genannt. Abb. 4 zeigt einen Graphen mit einem Knoten in der Mitte, der einen Graphen als Funktion aggregiert hat.

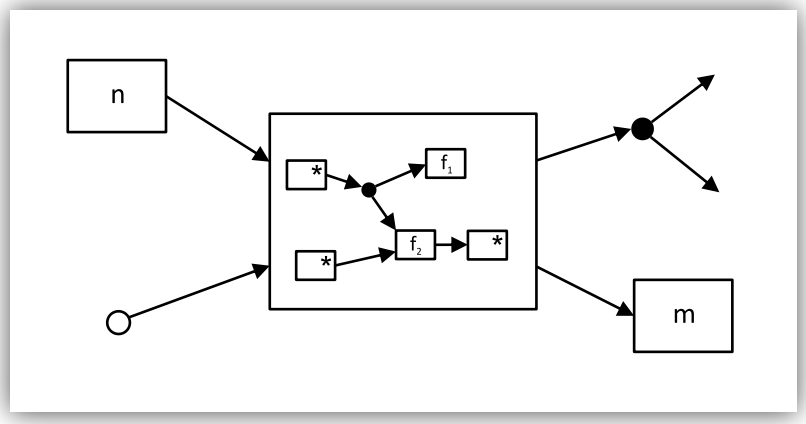


Abb. 4 Aggregation eines Graphen zu einem Knoten

Um das Verfahren zu beschreiben, werden einige Annahmen getroffen.

1. Funktionsknoten besitzen eine endliche Menge von Ein- und Ausgängen in Form von Anschlüssen.
2. Es gibt eine Teilmenge offener Ein- und Ausgänge an den Funktionsknoten des Sub-Graphen, welche für den Aggregationsknoten als Ein- und Ausgänge dienen sollen.

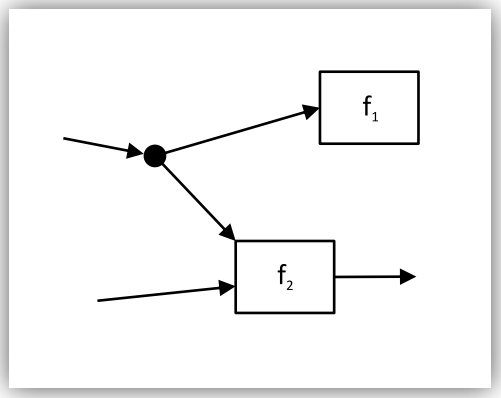


Abb. 5 Offene Ein- und Ausgänge in einem Flussgraphen

Abb. 5 zeigt einen Graphen mit offenen Ein- und Ausgängen.
Es wird nun eine besondere Art von Quellen und Senken eingeführt. Im Folgenden werden sie Super-Eingang und Super-Ausgang genannt. In den schematischen Darstellungen werden sie durch einen Stern gekennzeichnet.

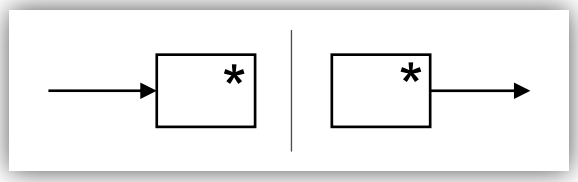


Abb. 6 (v.l.n.r.) Super-Aus- und -Eingang

Der Graph, der zu einem Knoten aggregiert werden soll, wird nachfolgend Sub-Graph genannt. Aufbauend auf den oben formulierten Annahmen kann der Sub-Graph nach folgendem Prinzip zu einem Knoten aggregiert werden:

Für jeden offenen Eingang an einem Funktionsknoten, der in dem Aggregationsknoten als Eingang dienen soll, wird ein Super-Eingang als Quelle in den Sub-Graphen eingefügt und eine Kante, welche von dem einzigen Ausgang des Super-Eingangs ausgeht und in dem offenen Eingang des Funktionsknotens endet.

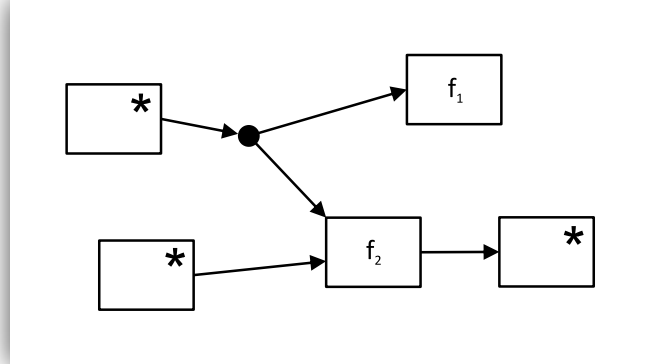


Abb. 7 Sub-Graph mit Super-Ein- und -Ausgängen

Für jeden offenen Ausgang an einem Funktionsknoten, der in dem Aggregationsknoten als Ausgang dienen soll, wird ein Super-Ausgang als Senke in den Sub-Graphen eingefügt und eine Kante, welche von dem offenen Ausgang des Funktionsknotens ausgeht und in dem einzigen Eingang des Super-Ausgangs endet.

Nun kann der Aggregationsknoten den Sub-Graphen verkapseln. Dazu muss er alle Marken, welche er an einem Eingang erhält, in den entsprechenden Super-Eingang des Sub-Graphen weiterleiten und alle Marken, welche von den Super-Ausgängen des Sub-Graphen konsumiert werden, an seinen entsprechenden Ausgang weiterleiten. Die Menge der Super-Ein- und -Ausgänge wird Schnittstelle eines Sub-Graphen genannt. Die Aktivierung des Aggregationsknotens folgt den Regeln des Datenflusssystems.

Mit diesem Konzept wird der Sub-Graph zur Funktion des Aggregationsknotens. Es wird dadurch möglich einen Flussgraphen als Komponente zu betrachten und eine fraktal verschachtelte Struktur von Flussgraphen zu entwerfen. Eine solche Struktur bietet gegenüber einem flachen Graphen einige Vorzüge. Das Flusssystem erhält die Fähigkeit der Komponierbarkeit. Aufbauend auf einer Bibliothek mit elementaren Funktionsknoten können komplexere Strukturen entwickelt werden, welche für eine höhere Abstraktionsebene wiederum als Entwicklungsbasis dienen können. Das schließt die Fähigkeit zur Abstraktion mit ein. Denn ein Entwickler, der einen Graphen auf einer höheren Abstraktionsebene entwickelt, muss sich nicht um die verborgene Komplexität in den Aggregationsknoten sorgen.

Das Prinzip der Kapselung, bzw. das Geheimnisprinzip kommt zur Anwendung. Ein Graphen-Entwickler auf einer höheren Abstraktionsschicht muss nicht wissen welche Funktionsknoten in den Sub-Graphen arbeiten und hat auch keine Möglichkeit deren Anschlüsse zu verwenden, wenn sie nicht als Super-Anschlüsse Teil der Schnittstelle des Sub-Graphen sind. Es wird möglich, Anforderungen an einen wiederverwendbaren Sub-Graphen, auf der Grundlage seiner Schnittstelle, zu definieren. Mit Hilfe der Schnittstelle und den definierten Anforderungen ist es relativ einfach, z.B. durch Black-Box-Tests, die Korrektheit eines Sub-Graphen zu verifizieren, womit eine Technik der Qualitätssicherung in der Flussgraphen-Entwicklung anwendbar wird.

2.2 Architektur von datenverarbeitenden Systemen

Es gibt verschiedene Ansätze die reine Rechenleistung eines Computers so zu abstrahieren, dass sie für den Menschen nutzbar wird. Die am weitesten verbreitete Rechnerarchitektur ist die Von-Neumann-Architektur, welche als SISD-Architektur in den heutigen PC-Prozessoren steckt. SISD bedeutet *Single Instruction Single Data* und ist die Grundlage vieler prozeduraler und objektorientierter Programmiersprachen. Eine weitere Architektur ist die in 2.1.2 bereits erwähnte Datenflussarchitektur. Solche und andere Architekturen lassen sich in verschiedenen Eigenschaften unterscheiden. Um sie nicht nur vergleichen, sondern auch klassifizieren zu können, möchte ich den Begriff des operationalen Modells (auch Ausführungsschema genannt) verwenden.

Als Grundlage für diesen Abschnitt diente (Mündemann, 1989).

Das operationale Modell einer Rechnerarchitektur beschreibt drei unterschiedliche Eigenschaften. Die erste Eigenschaft ist die Art wie die Operationen mit Daten versorgt werden, die zweite Eigenschaft ist das Modell der Befehlswirkung und die dritte Eigenschaft ist die Synchronisationsstrategie, mit der die Reihenfolge der Befehlsausführung definiert wird.

Für die Datenversorgung von Operationen gibt es drei Möglichkeiten (Treleaven, Brownbridge, & Hopkins, 1982): *by literal*, *by value* und *by reference*. Bei der ersten Möglichkeit *by literal* sind die Daten eines Operanden bereits zur Übersetzungszeit bekannt. In diesem Fall sind sie als Teil des Befehlscodes für die Operation direkt zugänglich. Die zweite Möglichkeit *by value* sieht vor, dass Ergebnisdaten einer Operation jeder nachfolgenden Operation als Kopie übergeben werden. Die letzte Möglichkeit *by reference* arbeitet mit Zeigern, welche auf die Daten zeigen. Operationen erhalten einen Zeiger auf die Daten als Operand. Die Daten werden für Berechnungen nicht kopiert.

Es gibt zwei Arten von Modellen für die Wirkung der Befehlsausführung (Britcher & Craig, 1986). Es gibt Flussmodelle (*flow model*) und Zustandsmodelle (*state model*). Das Kennzeichen der Flussmodelle ist, dass deren Operationen als Funktionen im mathematischen Sinne aufgefasst werden können. Dabei wird mit Hilfe eines Algorithmus‘ eine Menge von Eingangsdaten auf eine Menge von Ausgangsdaten abgebildet. Wird eine Operation mehrmals mit denselben Eingangsdaten ausgeführt, werden diese immer auf die gleichen Ausgangsdaten abgebildet. Bei den Zustandsmodellen besitzen die Operationen interne Zustandsdaten, welche während der Ausführung sowohl gelesen als auch geschrieben werden können. Dadurch kann eine wiederholte Ausführung der Operation mit denselben Eingangsdaten zu unterschiedlichen Ergebnissen führen.

Um die Reihenfolge der Befehlsausführung zu bestimmen, gibt es drei Ansätze (Treleaven, Brownbridge, & Hopkins, 1982), (Lima, Mundy, & Treleaven, 1983). Die sequentielle, die parallele und die rekursive Synchronisationsstrategie.

Die sequentielle Synchronisation arbeitet mit einer programmexternen globalen Instanz (z.B. einem Befehlszähler), welche die Reihenfolge und den Beginn der Befehlsausführung festlegt. Diese Synchronisationsstrategie wird auch Kontrollflussmodell genannt. Dabei kann die Befehlsausführung

vertikal durch Befehlsvorschau (*prefetching*) oder horizontal durch mehrere Ausführungspfade (*threads*) parallelisiert werden.

Bei der parallelen Synchronisation wird der Zeitpunkt für die Ausführung einer Operation anhand der Verfügbarkeit seiner Operanden ermittelt. Diese Strategie wird in der Literatur oft als *data driven computing* bezeichnet (Dennis, 1974). Dabei findet die Parallelisierung automatisch statt.

Die auch als *demand driven computing* bezeichnete rekursive Synchronisation basiert auf dem Ansatz, dass eine Operation ausgeführt wird, sobald ihre Ergebniswerte benötigt werden. Bei dieser Synchronisationsstrategie werden nur jene Operationen eines Programms abgearbeitet, die für das Ergebnis erforderlich sind. Um das Programm zu starten, wird eine Menge von Ergebniswerten bei einigen Operationen angefordert, welche ihre Operanden von anderen Operationen als Ergebniswerten anfordern. Dieser Anforderungsfluss setzt sich im Programm fort, bis die Operanden als Konstanten vorliegen und die Operationen nun in umgekehrter Reihenfolge ausgeführt werden können.

Die oben erläuterten drei Eigenschaften, welche ein operationales Modell charakterisieren, erlauben es nun, vorhandene oder neue Modelle zu klassifizieren. Dabei wird jedem Modell ein Tripel (D, B, S) mit

$$D \subseteq \{\text{"literal"}, \text{"value"}, \text{"reference"}\}$$
$$B \subseteq \{\textit{"flow"}, \textit{"state"}\}$$
$$S \subseteq \{\textit{"serial"}, \textit{"parallel"}, \textit{"recursive"}\}$$

zugeordnet.

Als Beispiele sind einige bekannte operationale Modelle mit ihrer Klassifizierung in Abb. 8 *aufgelistet.*

Modell	*Klassifikation*
Von-Neumann-Modell	$(\{\textit{"literal"}, \textit{"reference"}\}, \{\textit{"state"}\}, \{\textit{"serial"}\})$
Datenfluss-Modell	$(\{\textit{"literal"}, \textit{"value"}\}, \{\textit{"flow"}\}, \{\textit{"parallel"}\})$
Reduktionsmodell	$(\{\textit{"literal"}, \textit{"value"}\}, \{\textit{"flow"}\}, \{\textit{"parallel"}\})$

Abb. 8 Tabelle mit Klassifizierungen von operationalen Modellen

2.3 Objektorientierte Programmierung

Eng zusammen mit den operationalen Modellen, welche in 2.2 erläutert wurden, hängt das Programmierparadigma einer Programmiersprache. Dieser Abschnitt soll einen kurzen Überblick über das objektorientierte Programmieren geben.

2.3.1 Prozeduraler Ansatz

Bevor die Objektorientierung zur Strukturierung von Programmcode benutzt wurde, war das prozedurale Programmierparadigma der vorherrschende Ansatz. Bei diesem Ansatz gibt es einen Datenbereich und einen Programmbereich. Der Programmbereich besteht aus einer Menge von Prozeduren, welche üblicher Weise eine feste Adresse als Einsprungs-Punkt, eine Parameterliste und einen Rückgabetyp besitzen. Die Prozeduren haben die Möglichkeit sich gegenseitig aufzurufen und Daten als Parameter oder Rückgabewert auszutauschen. Typische Vertreter dieses Paradigmas sind FORTRAN, Cobol, C, Pascal und Basic.

Bei großen Programmen führt der prozedurale Ansatz dazu, dass sich der Quellcode des Programms immer schlechter verwalten lässt. Da jede Prozedur einen eindeutigen Namen benötigt, wurden in der Vergangenheit häufig kryptische Präfixe verwendet, um ähnliche Prozeduren aus unterschiedlichen Anwendungsdomänen oder Bibliotheken zu unterscheiden.

2.3.2 Geburt des Objektes

Bereits in den 1970er Jahren wurde mit Simula-67 eine Sprache entwickelt, die mit dem objektorientierten Ansatz andere Wege ging. Die Idee war, dass jene Quellcodeabschnitte, welche zum Verarbeiten einer bestimmten Gruppe von Daten vorgesehen waren, mit diesen Daten zusammengefasst werden sollten. Ein Objekt ist in diesem Sinne eine Kapsel, die eine Anzahl strukturierter Daten und jene Quellcode-Abschnitte enthält, welche zum Verarbeiten dieser Daten vorgesehen sind. Ausgehend von diesem Grundgedanken wurden einige Konzepte als Teil der Objektorientierung entwickelt.

2.3.3 Merkmale der Objektorientierung

Eine ***Klasse*** ist die Vorlage für ein Objekt. Sie beschreibt die Datenstrukturen, auch ***Felder*** genannt, die ein Objekt dieser Klasse kapseln soll und enthält die Quellcodeabschnitte zur Verarbeitung der gekapselten Daten. Diese Quellcodeabschnitte werden ***Methoden*** genannt und ähneln vom Aufbau einer Prozedur, besitzen jedoch zur Ausführungszeit immer den Kontext eines Objektes. D.h. dass Methoden nur für eine konkrete Instanz einer Klasse, also einem Objekt, aufgerufen werden können. Beim Zugriff auf die Daten und Methoden eines Objektes spielt der Begriff ***Datenkapselung*** oder ***Geheimnisprinzip*** eine wichtige Rolle. Denn für jedes Feld und jede Methode wird definiert, ob der Zugriff von außerhalb des Objektes möglich ist. Die Menge aller öffentlichen Methoden und Felder bildet dabei eine Schnittstelle für das Objekt. Diese Schnittstelle ermöglicht die Kommunikation mit anderen Objekten. Alle Implementierungsdetails über die Verarbeitung der Daten werden im Objekt verborgen. Ein weiteres wichtiges Merkmal der Objektorientierung ist die ***Vererbung***. Klassen können ihre Fähigkeiten (Felder und Methoden) an Unterklassen vererben. Dabei erhält die Unterklasse automatisch alle Fähigkeiten der Oberklasse und kann diese durch eigene Felder und Methoden erweitern. Der Begriff ***Polymorphie*** wird verwendet, wenn zwei Objekte bei Aufruf derselben Methode unterschiedlich reagieren. Dieses Verhalten kann durch das ***Überschreiben*** von Methoden hervorgerufen werden. Dabei erben z.B. zwei Unterklassen dieselbe Methode von einer gemeinsamen Oberklasse. Jedoch deklarieren beide Unterklassen eine eigene Methode unter dem gleichen Bezeichner,

welche jedoch andere Befehle ausführt als die Methode der Oberklasse. Wird zur Laufzeit die Methode mit dem Bezeichner der Oberklasse aufgerufen, wird durch ***späte Bindung*** ermittelt, ob für die Instanz, auf welcher der Aufruf erfolgt, die Methode mit Hilfe einer Unterklasse überschrieben wurde. Ist dies der Fall, wird die Methode der Unterklasse an Stelle der Oberklassen-Methode ausgeführt.

Aktuelle objektorientierte Programmiersprachen sind z.B. Java, Visual Basic 8, C++, C# und D.

2.3.4 Statische Elemente

Die meisten objektorientierten Sprachen bieten die Möglichkeit, von dem reinen Konzept abzuweichen, und in den objektorientierten Code sog. statische Elemente einzubringen. Als ***statisch*** werden in diesem Kontext alle jene Felder und Methoden bezeichnet, auf die ohne ein Objekt zugegriffen werden kann. Statische Felder existieren im gesamten Programm nur einmal. Im Gegensatz zu dynamischen Feldern, welche für jedes Objekt einmal existieren. Statische Methoden besitzen keinen Objektkontext und können damit nur auf andere statische Felder und Methoden zugreifen.

Statische Elemente sind eine Form der Abwärtskompatibilität zur prozeduralen Programmierung. Da C++ als Erweiterung von C betrachtet werden kann, besitzt C++ automatisch die prozeduralen oder statischen Anteile von C. In C# und Java wurde die Möglichkeit geschaffen innerhalb einer Klasse statische Felder und Methoden zu deklarieren, welche vom Verhalten dem prozeduralen Programmierparadigma entsprechen.

2.3.5 Namensräume und Pakete

Bei der Entwicklung von Anwendungen ist es häufig der Fall, dass mehrere Klassen mit den gleichen Bezeichnern benannt werden. Z.B. ist es sinnvoll eine Klasse, deren Objekte Dateien einlesen können, `Reader` zu nennen. Genauso sinnvoll ist es aber eine Klasse, deren Objekte Tastatureingaben lesen können, `Reader` zu nennen. In diesem Fall ließe sich der entstandene Namenskonflikt recht einfach lösen, indem man die erste Klasse `FileReader` und die zweite `KeyboardReader` nennt. Es gibt jedoch reichlich Fälle in denen der Unterschied der Klassen zu sehr langen oder unleserlichen Namen führen würde. Um solche und ähnliche Namenskonflikte zu lösen, gibt es in den meisten objektorientierten Sprachen eine Form von Namensräumen. In diesem Konzept wird jede Klasse einem Namensraum zugeordnet. Innerhalb eines Namensraums müssen die Bezeichner der enthaltenen Elemente eindeutig sein und die Namensräume benötigen einen nach Möglichkeit weltweit eindeutigen Bezeichner. Um eine Klasse absolut zu benennen, wird ihrem Namen der Namensraum mit einem Trennzeichen vorangestellt. So könnte man den Namenskonflikt im obigen Beispiel auflösen, indem man die beiden Klassen unterschiedlichen Namensräumen zuordnet. Die erste Klasse könnte einem Namensraum `FileSystem` zugeordnet werden und die zweite Klasse einem Namensraum `UserInterface`. Absolut könnte nun die erste Klasse mit `FileSystem::Reader` und die zweite Klasse mit `UserInterface::Reader` benannt werden.

Namensräume haben jedoch nicht nur den Zweck Namenskonflikte zu vermeiden. Sie dienen vielmehr dazu, eine große Anzahl von Klassen oder anderen Code-Elementen semantisch zu gruppieren. In diesem Kontext wird ein Namensraum auch als ***Paket*** bezeichnet. Pakete können Elemente enthalten, die außerhalb des Pakets nicht sichtbar sind. Ein Paket kann folglich in Form aller Klassen und Code-Elemente, welche außerhalb des Pakets sichtbar sind, eine ähnliche Art von Schnittstellen anbieten wie Klassen.

In manchen Fällen ist es sinnvoll, nur eine Klasse des Pakets öffentlich sichtbar zu kennzeichnen. Diese Klasse wird Fassade genannt und bietet Zugriff auf die Funktionalität des Paketes. Mit einer Fassade können nach dem Geheimnisprinzip die Implementierungsdetails eines ganzen Paketes verborgen werden.

Wichtig für die Software-Architektur ist das Definieren und Überprüfen von Abhängigkeiten zwischen Paketen. Ziel ist es Klassen und Code-Elemente so in Pakete zu gruppieren, dass die Kohäsion innerhalb eines Paketes möglichst groß, zwischen den Paketen jedoch möglichst klein ist. Die Abhängigkeiten zwischen Paketen sollten keine Kreise bilden.

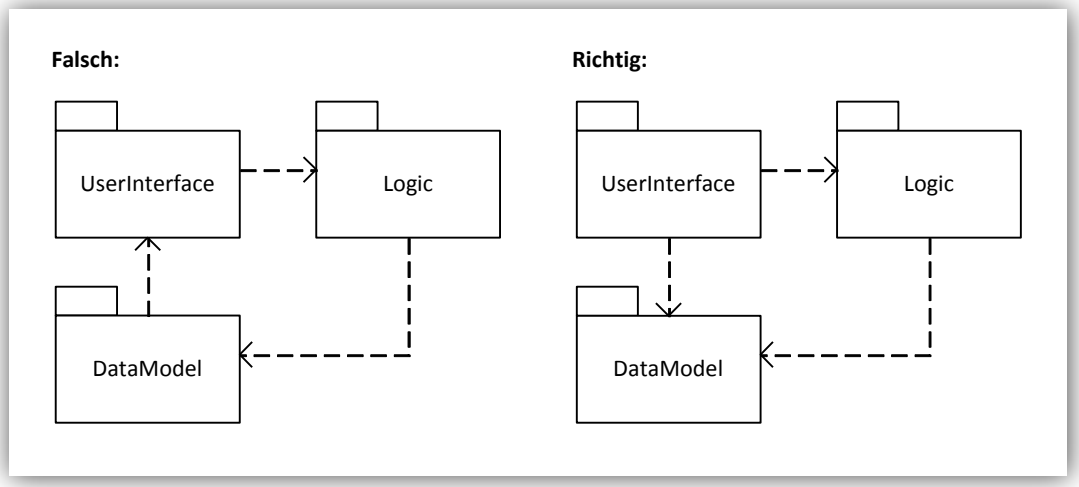


Abb. 9 Abhängigkeiten von Paketen

2.4 Microsoft .NET und das Mono-Projekt

Am Anfang sollte die Definition von Microsoft vorangestellt werden:

„Das .NET Framework ist die Umgebung für das Programmieren, Bereitstellen und Ausführen von XML-Webdiensten und anderen Anwendungen. Die beiden zentralen Komponenten sind die CLR-Umgebung (Common Language Runtime) sowie die umfangreichen Klassenbibliotheken.“

(Microsoft Deutschland, 2004)

2.4.1 Microsoft .NET-Framework

Der Begriff .NET (*lies dott-nett*) ist zunächst der Name für die Weiterentwicklung des Komponentenmodells COM+. Und doch steht noch viel mehr hinter dem Namen als nur ein Komponentenmodell. Das .NET-Framework umfasst eine Vielzahl von Technologien. Einen sehr guten Einblick verschafft (Schäpers, Huttary, & Bremes, 2002) worin im Kapitel 1 und 2 das Konzept hinter .NET beschreiben wird.

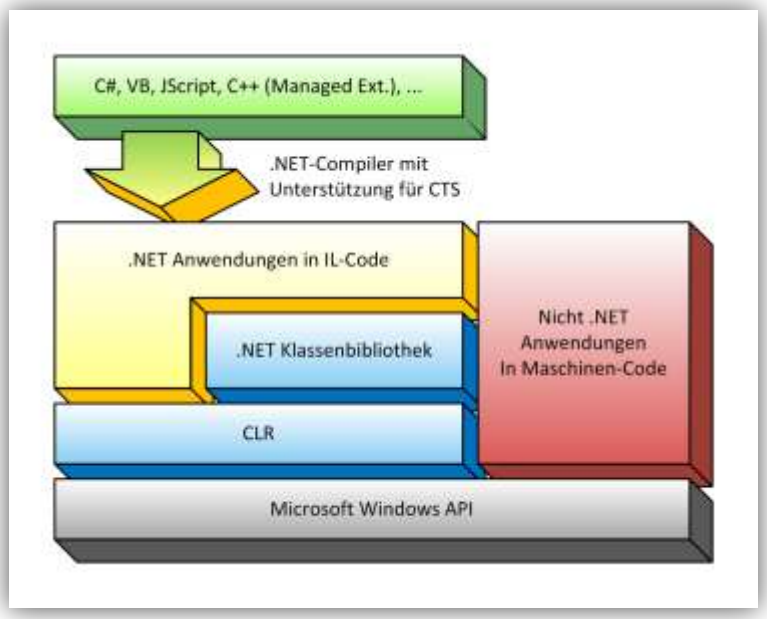


Abb. 10 Übersicht über den Aufbau von .NET

An dieser Stelle soll kurz auf den Aufbau und die Hauptbestandteile des Frameworks eingegangen werden, wobei das oben genannte Buch als Grundlage dient.

Erklärtes Ziel von Microsoft für das .NET-Framework war das Komponentenmodell so zu gestalten, dass aus einer großen Anzahl von Sprachen Komponenten kompiliert werden können, welche direkt kompatibel sind und sich gegenseitig aufrufen können.

Wenn man die Unterschiede der verschiedenen Sprachen betrachtet, stellt man schnell fest, dass die größten Probleme die unterschiedlichen Datentypen und die Unterschiede in den konkreten Vererbungsmodellen darstellen. Wenn z.B. eine Komponente, welche in C++ implementiert wurde, eine Methode aus einer Visual Basic-Komponente aufruft, ist es nicht einfach die Datentypen der Parameter so zu wählen, dass beide Sprachen das Gleiche darunter verstehen.

Die Basis für die Kompatibilität der verschiedenen Sprachen in .NET ist deshalb ein zentrales Typen-System oder ***Common Type System*** (CTS). Es ist eine Spezifikation, der alle .NET-Sprachen unterworfen sind. Das CTS definiert die .NET-Basisklassen (`System.Object`, `System.Int32`, `System.UInt16`, etc.) und enthält auch die sprachspezifischen Datentypen. Viele .NET-Sprachen nutzen die in dem CTS implementierten Basis-Datentypen jedoch unter einem sprachspezifischen Namen. So ist der Werttyp `System.Int32` in C# unter dem Namen `int` verfügbar. Daneben beschreibt das CTS z.B. die Form der Vererbung.

Im Anhang D - Diagramm 1 befindet sich eine Übersicht aus (Schäpers, Huttary, & Bremes, 2002) Seite 38, die sehr schön veranschaulicht, welche verschiedenartigen Datentypen das CTS vorsieht.

Jede .NET-Sprache wird von einem .NET-Compiler nicht in plattformspezifischen Maschinencode übersetzt, sondern in eine sog. ***Intermediate Language*** (IL). Diese Sprache ist der gemeinsame Nenner aller .NET-Sprachen und, wie weiter unten deutlich wird, auch aller unterstützenden Plattformen. Das CTS bezieht sich mit seinen Spezifikationen auf die IL und definiert alle Basis-Datentypen in der IL. Eine ausführbare Einheit, welche für gewöhnlich als DLL- oder EXE-Datei vorliegt, wird ***Assembly*** genannt. Sie wird aus einer Kompiliereinheit gebildet und kann aus beliebig vielen Quellcode-Dateien entstehen. (In Microsoft Visual Studio erzeugt jedes Projekt ein *Assembly*.) Ein *Assembly* enthält im Falle einer DLL eine Sammlung von Klassen und im Falle einer EXE zusätzlichen einen statischen Einsprungspunkt für den Aufruf durch die CLR beim Starten des .NET-Programms.

Damit Komponenten unterschiedlicher Sprachen nun reibungslos kommunizieren können, ist ein kleinster gemeinsamer Nenner der Datentypen erforderlich. Denn das CTS lässt durchaus auch sprachspezifische Datentypen zu. Diesen stellt die ***Common Language Specification*** (CLS) dar. Sie bildet eine Untermenge der CTS-Datentypen, welche von jeder .NET-Sprache unterstützt werden muss. Sie ist ausschließlich objektorientiert.

Quellcode in den unterschiedlichen Sprachen darf in öffentlichen Methoden und Feldern nur CLS-konforme Datentypen verwenden, damit der Austausch der Daten zwischen Komponenten unterschiedlicher Sprachen reibungslos verläuft.

Auf der Grundlage der CLS hat Microsoft eine umfangreiche Klassenbibliothek entwickelt, welche die Basis für .NET-Anwendungen darstellt. Das Spektrum dieser ***.NET Class Library*** reicht von einfachen IO-Operationen für Dateisystem und Netzwerk über XML-Verarbeitung, Unterstützung für verteilte Anwendungen (***.NET Remoting***), Oberflächen- und Grafik-Schnittstellen (***Windows-Forms***) bis hin zu Webdiensten. Der Umfang von aktuellen Büchern über .NET oder eine .NET-Sprache lässt deutlich werden, dass es viel zu sagen gibt über die *.NET Class Library*. Wobei auch diese Bücher oftmals nur an der Oberfläche kratzen können.

Zur Ausführung von .NET-Programmen ist eine Laufzeitumgebung erforderlich. Diese heißt ***Common Language Runtime*** (CLR) und erfüllt eine Vielzahl von Aufgaben. Die folgende Übersicht aus (Schäpers, Huttary, & Bremes, 2002) Seite 42 veranschaulicht die verschiedenen Bereiche der CLR.

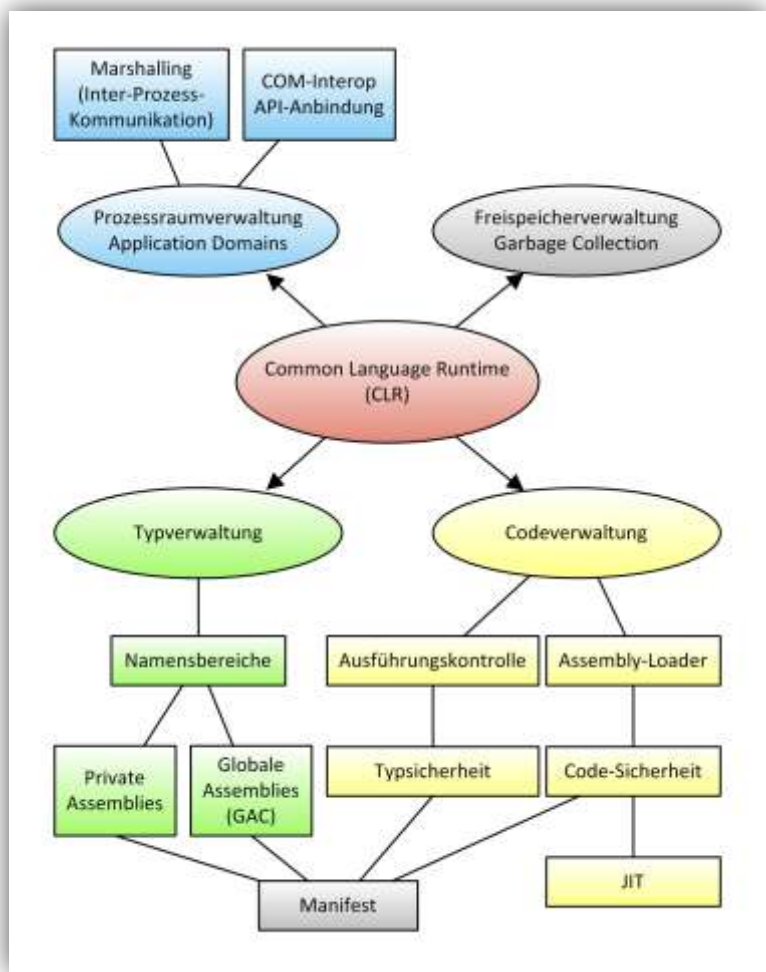


Abb. 11 Aufgaben der CLR

Eine ***Application Domain*** ist eine Art Prozessraum mit der Besonderheit, dass der Prozess eines .NET-Programms mehrere *Application Domains* enthalten kann. Zwischen *Application Domains* sind keine direkten Methoden-Aufrufe möglich. Statische Felder existieren in jeder *Application Domain* einmal. Für die Kommunikation zwischen *Application Domains* ist *.NET-Remoting* nötig.

Hervorzuheben ist auch der ***Garbage Collector*** (GC) der CLR, welcher für die automatische und asynchrone Freigabe von nicht mehr benötigtem Speicher zuständig ist.

Desweiteren ist der *Assembly Loader* interessant, der es gestattet zur Laufzeit weitere Assemblies in eine *Application Domain* zu laden und auszuführen.

Der ***Just-In-Time-Compiler*** (JIT) der CLR ist bei der Ausführung eines .NET-Programms dafür zuständig, den IL-Code in echte Maschinenbefehle zu übersetzten. Das ist für globale Assemblies nicht mehr notwendig, weil diese bei der Installation im ***Global Assembly Cache*** bereits komplett in Maschinensprache übersetzt worden sind.

2.4.2 Das Mono-Projekt

Auf der Website von Mono[1] steht folgende kurze Beschreibung:

"Mono provides the necessary software to develop and run .NET client and server applications on Linux, Solaris, Mac OS X, Windows, and UNIX. Sponsored by Novell, the Mono open source project has an active and enthusiastic contributing community and is positioned to become the leading choice for development of Linux applications."

Das Mono-Projekt hat sich zur Aufgabe gemacht das .NET-Framework für verschiedene Plattformen zu implementieren. Dies ist möglich, weil Microsoft sowohl die Spezifikation für das .NET-Framework als auch die Programmiersprache C# standardisieren ließ. Auf der Website von Mono (ECMA - Mono) sind die entsprechenden Standards aufgeführt:

- Ecma-334 C# Specification
  http://www.go-mono.com/ecma/Ecma-334.pdf
- Ecma-335 CLI Specification – Virtual Machine
  http://www.go-mono.com/ecma/Ecma-335.pdf
- Ecma-335 CLI Specification – XML-based class library
  http://www.go-mono.com/ecma/Ecma-335-xml.zip
- TR-84 CLI Technical Report – Microsoft© Word and PDF class library specification
  http://www.go-mono.com/ecma/TR-084.zip
- TR-89 CLI Technical Report – Additional Generics Library
  http://www.go-mono.com/ecma/TR-089.pdf

Wie weit dieses Bestreben inzwischen gediehen ist, bezeugt der *Mono Migration Analyzer*, dem ein oder mehrere Assemblies übergeben werden. Er überprüft die Referenzen zum .NET-Framework und gibt an, falls eine Methode oder Klasse in Mono noch nicht oder nur teilweise implementiert ist.

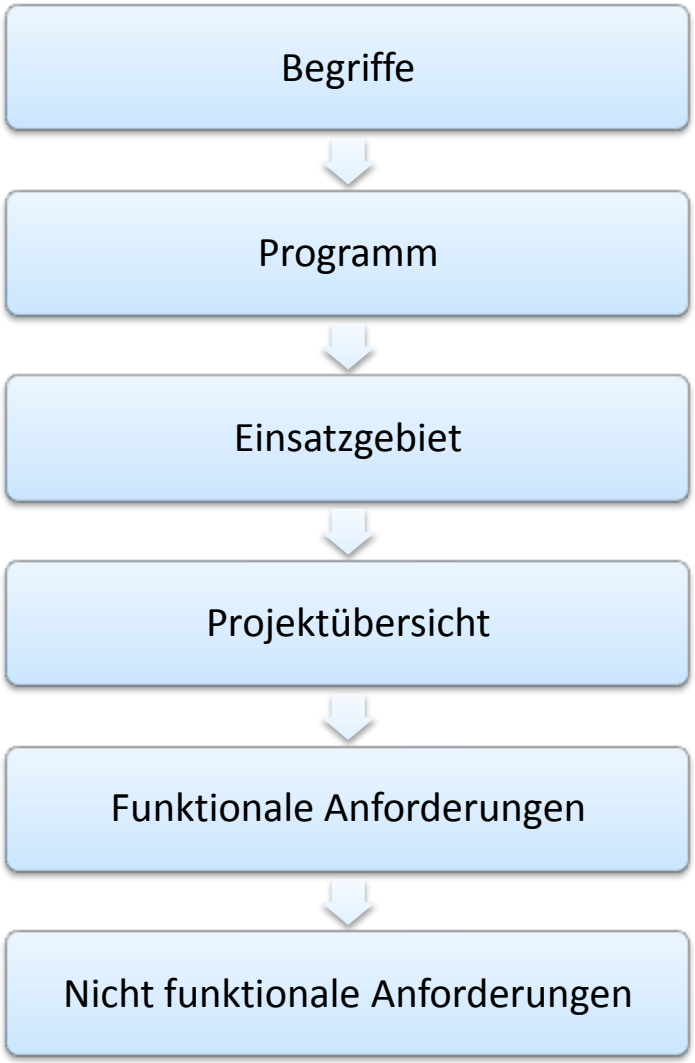
Aus den Vergleichsdaten des *Mono Migration Analyzers* für die aktuelle Mono-Version 1.2.4 geht hervor, dass noch 10328 Methoden aus dem Microsoft .NET in Mono fehlen. Bei 3771 Methoden wird eine Ausnahme geworfen und 3733 Methoden stehen auf der ToDo-Liste für die Weiterentwicklung. Als .NET-Entwickler hat man im *Mono Migration Analyzer* die Möglichkeit dem Mono-Entwickler-Team mitzuteilen, welche Methoden bei der Weiterentwicklung priorisiert werden sollen.

Nach Angaben der Website ist es dennoch mit teilweise geringem Portierungsaufwand bereits jetzt möglich, größere Projekte auf Mono zu portieren und auszuführen. Dabei kann eine in Microsoft Visual Studio kompilierte .NET-EXE oder -DLL z.B. auf einem Linux Rechner mit Mono ausgeführt werden, solange sie ausschließlich die von Mono implementierten Teile der *.NET Class Library* verwendet.

[1] http://www.mono-project.com

3 Spezifikation

Dieser Abschnitt enthält die Spezifikation für das Programmiersystem DynamicNodes. Dafür werden zunächst einige Begriffe eingeführt und kurz erläutert. Anschließend wird der Aufbau eines DynamicNode-Programms beschrieben. Dann folgen eine Eingrenzung des Einsatzgebietes und eine kurze Projektübersicht. Im Anschluss daran enthalten die Kapitel 3.5 und 3.6 die funktionalen und nicht funktionalen Anforderungen an DynamicNodes.



3.1 Begriffe

- Programm
  Ein Programm ist in DynamicNodes ein gerichteter Flussgraph. Ein Programm besteht aus Operationsknoten und Verbindungen.
- Operationsknoten
  Ein Operationsknoten ist ein Knoten in einem Programm. Er besitzt eine endliche Menge an Ein- und Ausgängen.
- Operation
  Die Operation eines Operationsknotens ist der Mechanismus, welcher die eingehenden Marken verarbeitet und ausgehende Marken erzeugt.
- Kontentyp
  Es gibt unterschiedliche Operationsknoten. Sie unterscheiden sich durch die Anzahl von Ein- und Ausgängen und durch die Operation. Eine Klasse von gleichen Operationsknoten wird als Knotentyp bezeichnet.
- Operationsknotenbibliothek
  Eine Operationsknotenbibliothek ist eine Sammlung von Knotentypen.
- Eingang
  Ein Eingang ist eine Verbindungsmöglichkeit an einem Knoten. An einem Eingang kann eine eingehende Verbindung enden. Ein Eingang nimmt Marken von einer Verbindung entgegen.
- Ausgang
  Ein Ausgang ist eine Verbindungsmöglichkeit an einem Knoten. An einem Ausgang können ausgehende Verbindungen beginnen. Ein Ausgang sendet Marken über Verbindungen aus.
- Anschluss
  Anschluss ist der Oberbegriff für Eingang und Ausgang.
- Verbindung
  Eine Verbindung beginnt in genau einem Ausgang und endet in genau einem Eingang. Sie transportiert Marken von einem Ausgang zu einem Eingang.
- Marke
  Eine Marke ist ein Container, welcher die Informationen transportiert, die von einem Programm verarbeitet werden.
- Knotenreferenz
  Jeder Knotentyp besitzt eine für den Benutzer verständliche Beschreibung seiner Operation und seiner Anschlüsse, darin können Anwendungsbeispiele und Hinweise enthalten sein. Die Sammlung der Beschreibungen aller Knotentypen in einem DynamicNode-System wird Knotenreferenz genannt.

3.2 Programm

Ein Programm in DynamicNodes ist ein Flussgraph, welcher aus Operationsknoten und Verbindungen besteht. Ein Operationsknoten wiederum besteht aus Anschlüssen und einer Operation.

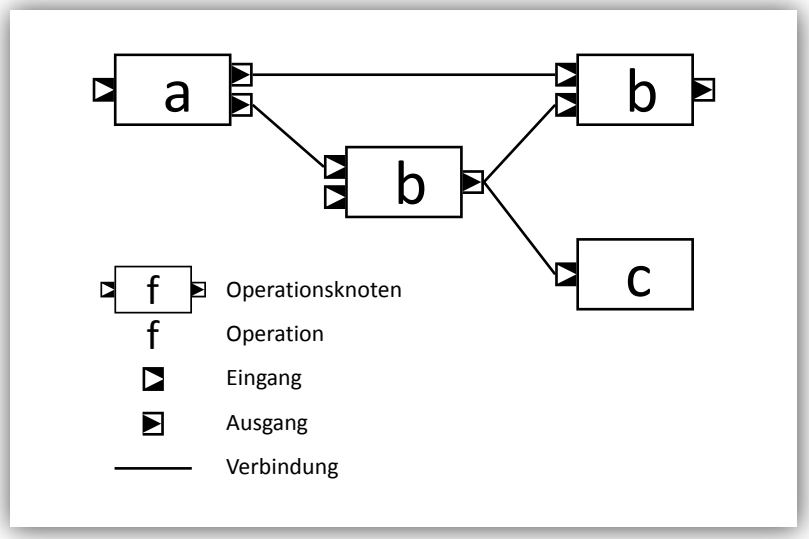


Abb. 12 DynamicNodes-Programm

Die Operation *f* eines Knotens wird durch eine Methode im objektorientierten Sinne repräsentiert. Diese Methode muss in einer verbreiteten imperativen Programmiersprache verfasst werden können, welche das objektorientierte Programmierparadigma unterstützt. Sie hat Zugang zu einer Schnittstelle, welche das Lesen von eingehenden Marken und das Erzeugen von ausgehenden Marken des Knotens ermöglicht.

Operationen können, müssen jedoch nicht zustandsbehaftet sein. Somit kann *ein* Programm sowohl reine Flussknoten, als auch Knoten enthalten, welche eine Zustandsmaschine enthalten bzw. darstellen.

Die Information, welche von einer Marke gekapselt wird, heißt auch Markenwert. Die Menge D aus der operationalen Modellklassifikation in 2.3 führt drei Klassen von Werttypen ein: zur Entwurfszeit festgelegte und konstante Informationen („by literal“), zur Laufzeit berechnete Informationen („by value“) und einen Verweis auf Informationen („by reference“). Jede der drei Klassen enthält eine Vielzahl von Datentypen. Marken müssen Werte mit Datentypen aus allen drei Klassen aufnehmen können.

Ein Anschluss kann eine endliche Menge an Datentypen unterstützen. Für jeden Anschluss muss definiert sein, welche Datentypen er unterstützt. Jeder Knotentyp besitzt eine Reihe von *Metadaten*:

- Eine über Operationsknotenbibliotheken hinweg eindeutige ID des Knotentyps
- Ein Name, welcher die Operation des Kontentyps in einem oder zwei Worten beschreibt. Optional kann der Name in unterschiedlichen Sprachen vorliegen, damit Knotentypen von Benutzern mit unterschiedlichen Sprachen leicht identifiziert werden können.

- Eine Kurzbeschreibung in Form von einem oder zwei Sätzen, welche die Operation des Knotentyps ausführlicher beschreibt. Auch dieser Beschreibungstext kann optional in mehreren Sprachen vorliegen.
- Die ausführliche Beschreibung des Knotentyps, seiner Operation und seiner Anschlüsse, welche in die Knotenreferenz eingegliedert wird.

Es muss einen Operationsknoten geben, welcher ein Programm als Operation ausführt. Dabei soll als Grundlage das in 2.1.3 vorgestellte Verfahren für die „Aggregation von Graphen zu Knoten“ zur Anwendung kommen.

3.3 Einsatzgebiet

Für die Anwendung von DynamicNodes sind drei Einsatzszenarien vorgesehen:

a) Experimentierbaukasten für den fachfremden Benutzer
b) Experimentier- und Programmierumgebung für den Fachmann
c) Laufzeitumgebung für fertige Flussgraphen in produktionsnahem Umfeld

Zusätzlich kommt DynamicNodes in der Entwicklungsphase von Operationsknoten als Testumgebung zum Einsatz.

3.4 Projektübersicht

DynamicNodes besteht im Wesentlichen aus einer Laufzeitumgebung, einem graphischen Editor, einer Testumgebung für Operationsknoten und einem Hilfesystem.

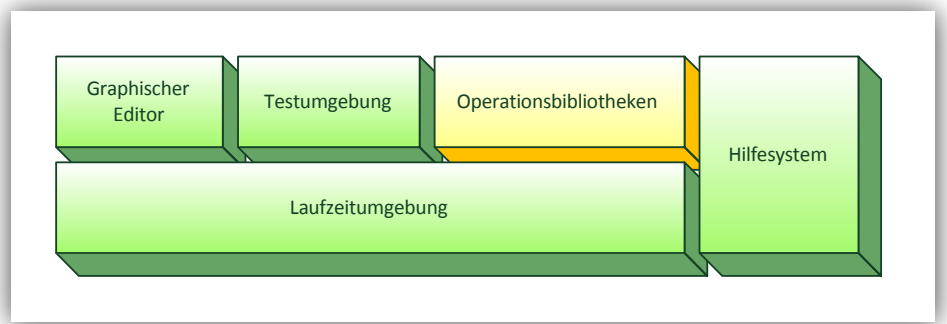


Abb. 13 Projektübersicht

Die Laufzeitumgebung ist sowohl für die Ausführung als auch für die Persistenz der Programme zuständig. Der graphische Editor ist der wichtigste Teil der Benutzerschnittstelle. Er ermöglicht das Erstellen und Bearbeiten von Programmen. Die Testumgebung hat zur Aufgabe, die Lauffähigkeit von Operationsknoten in der DynamicNodes-Laufzeitumgebung zu überprüfen. Sie ist ein wichtiges Werkzeug für die Entwicklung von Operationsknoten durch Dritte. Zu einer sinnvollen Installation von DynamicNodes gehört mindestens eine Operationsknotenbibliothek. Das Hilfesystem ist die erste Anlaufstelle für Fragen über die Funktionsweise von DynamicNodes und es stellt die Knotenrefe-

renz zur Verfügung. Das Hilfesystem soll den Einstieg in die Arbeit mit DynamicNodes vereinfachen und während der Arbeit als Nachschlagewerk dienen.

3.5 Funktionale Anforderungen

In diesem Abschnitt werden für jeden Programmteil erforderliche Eigenschaften und Funktionen beschrieben.

3.5.1 Laufzeitumgebung

Die DynamicNodes-Laufzeitumgebung muss Programme ausführen können. Sie muss fähig sein sowohl mehr als einen Operationsknoten als auch mehr als ein Programm parallel auszuführen. Dabei muss sie (wenn vorhanden) auf mehrere Prozessoren (bzw. Prozessorkerne) skalieren. Skalieren bedeutet in diesem Kontext dass die Operation jedes aktivierten Knoten auf einem separaten Prozessor ausgeführt wird. Das ist nur möglich, solange die Anzahl der aktivierten Knoten kleiner oder gleich der Anzahl der Prozessoren ist. Wenn die Anzahl der aktivierten Knoten zu einem Zeitpunkt die Anzahl der Prozessoren übersteigt, sollte die Rechenleistung der Prozessoren unter den Knoten aufgeteilt werden.

Die Laufzeitumgebung muss auf einem PC-System mit einem oder mehreren x86 kompatiblen Prozessoren mit der Von-Neumann-Architektur lauffähig sein. Sie sollte auf einem Microsoft Windows Betriebssystem und optional auch auf Linux oder weiteren Betriebssystemen ausgeführt werden können. Die Laufzeitumgebung muss die Datenstruktur der Programme im Arbeitsspeicher verwalten.

Die Laufzeitumgebung muss weder einen Debug-Modus besitzen, noch das explizite Starten, Pausieren oder Stoppen der Ausführung eines Programms unterstützen. Sobald ein Graph mindestens einen Knoten enthält, wird er ausgeführt. Sobald ein Programm aus der Laufzeitumgebung entfernt wird, endet auch die Ausführung.

Die Laufzeitumgebung muss eine Schnittstelle anbieten, welche folgende Fähigkeiten enthält:

- Programme für die Ausführung aufnehmen oder freigeben
- Programme von der Festplatte laden und auf der Festplatte speichern
- Ausführungsparameter der Laufzeitumgebung lesen und schreiben
- Über Änderungen an der Struktur der geladenen Programme informieren

Ein Teil der Laufzeitumgebung ist die Fähigkeit Programme zu speichern und zu rekonstruieren. Für die Speicherung der Programme muss eine Form gewählt werden, die eine leichte Weitergabe der Programme ermöglicht.

Die Datenstruktur eines Programms muss eine Möglichkeit bieten, zusätzliche Daten aufzunehmen, welche für die eigentliche Ausführung des Programms nicht erforderlich sind. Dazu zählen z.B. visuelle Eigenschaften von Operationsknoten und Verbindungen. Aus diesem Grund muss das Speicher-

format die Fähigkeit besitzen, generisch zusätzliche Daten zu speichern, deren Datentypen zur Entwurfszeit der Laufzeitumgebung noch unbekannt sind.

Marken an offenen Anschlüssen mit primitiven Datentypen müssen mit dem Programm gespeichert werden. Unter primitiven Datentypen sind Wahrheitswerte, Zahlen und Zeichenfolgen zu verstehen. Für Marken mit komplexen Datentypen ist diese Anforderung nicht zwingend. Komplexe Datentypen sind in diesem Kontext z.B. Klassen im objektorientierten Sinn.

Die Laufzeitumgebung muss während ihrer Initialisierung eine beliebige Anzahl von Operationsknotenbibliotheken laden können. Die Initialisierung der Laufzeitumgebung muss abgeschlossen sein, bevor ein Programm ausgeführt werden kann. Programme können nur korrekt rekonstruiert und ausgeführt werden, wenn die notwendigen Operationsknotenbibliotheken in die Laufzeitumgebung geladen wurden. Notwendig ist eine Operationsknotenbibliothek für ein Programm, wenn das Programm eine oder mehrere Instanzen eines Knotentyps enthält, welcher durch die Operationsknotenbibliothek definiert ist.

3.5.2 Editor

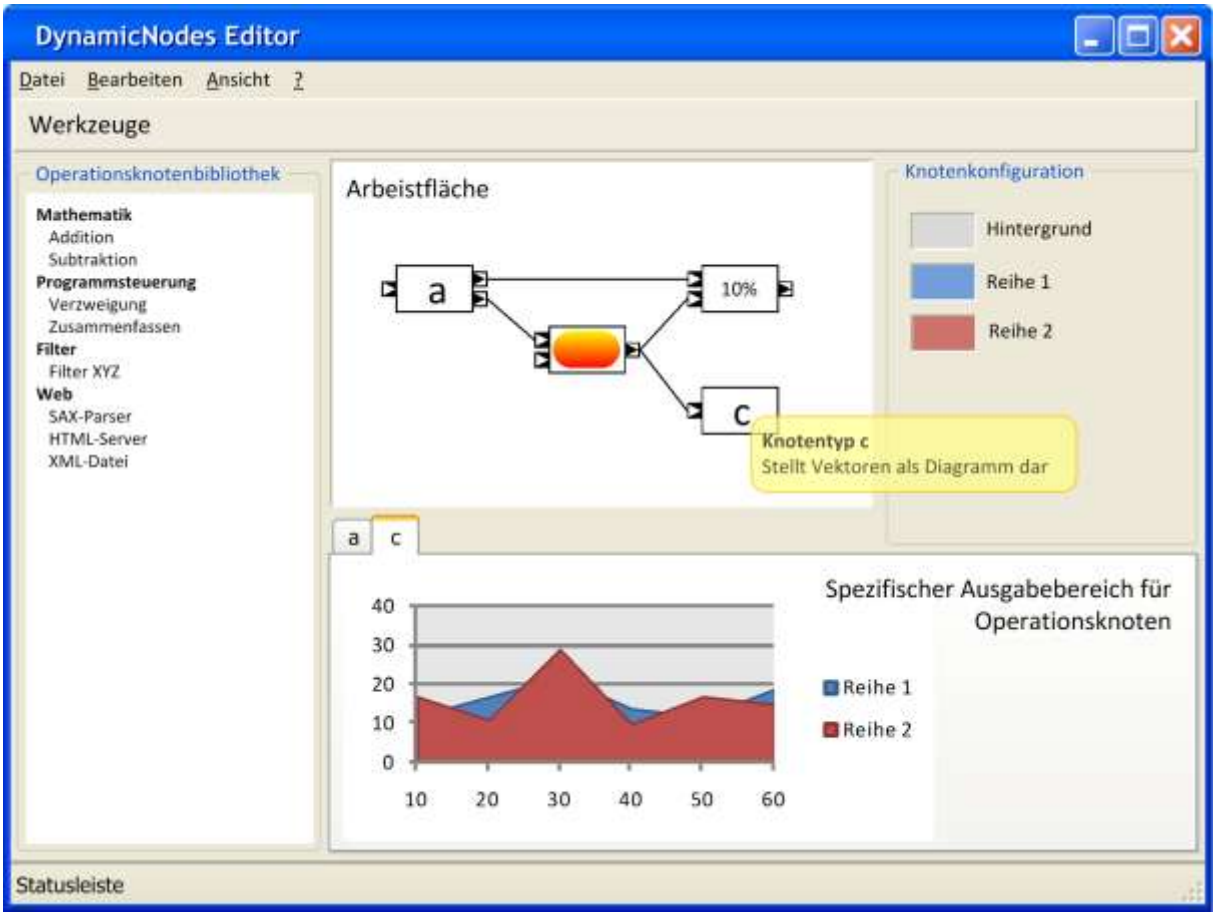


Abb. 14 Vorschlag für das Layout des Editors

Teil von DynamicNodes ist ein grafischer Editor zum Erstellen und Bearbeiten von Programmen. In Abb. 14 wird ein Vorschlag für ein mögliches Layout der Benutzerschnittstelle des Editors gemacht.

Die farbigen Details der Abbildung haben lediglich die Aufgabe die Phantasie des Lesers anzuregen und besitzen darüber hinaus keine Bedeutung.

Der Editor muss Knoten und Verbindungen eines Programms in einer geeigneten Weise zweidimensional auf dem Bildschirm anzeigen und folgende Operationen durch Tastatur- und Mausbefehle für den Benutzer zur Verfügung stellen:

- Auf der Festplatte gespeicherte Programme öffnen und in die Laufzeitumgebung laden
- Programme auf die Festplatte speichern
- Programme schließen und aus der Laufzeitumgebung entfernen
- Alle aus den Operationsknotenbibliotheken geladenen Knotentypen zur Auswahl anzeigen
- Eine Instanz eines Knotentyps erzeugen und in ein Programm einfügen
- Einen Operationsknoten aus einem Programm entfernen und zerstören
- Verbindungen unter Berücksichtigung der Datentyp-Kompatibilität von Anschlüssen erstellen
- Verbindungen löschen
- Operationsknoten eines Programms auf einer zweidimensionalen Arbeitsfläche frei anordnen
- Wenn von einem Operationsknoten gefordert, eine spezifische Benutzerschnittstelle zur Konfiguration des Knotens anzeigen
- Vergrößern und verkleinern der grafischen Ansicht eines Programms (Zoomen)
- Erzeugen von Marken an offenen Eingängen, durch Eingabe von Informationen mit primitiven Datentypen und einer ausgewählten Anzahl komplexer Datentypen
- Anzeigen der anliegenden Marken(werte) an Ein- und Ausgängen
- Anzeigen der Metadaten für einen Operationsknoten bzw. seinen Knotentyp
- Konfiguration der Laufzeitumgebung mit Hilfe eines Einstellungsdialoges

Dem Benutzer muss auf intuitive Weise deutlich werden, welche Eingänge zu welchen Ausgängen kompatibel sind, d.h. von welchem Ausgang, zu welchem Eingang eine Verbindung aufgebaut werden kann. Der Benutzer muss daran gehindert werden, dass er ungültige Verbindungen erzeugt.

Ungültig ist eine Verbindung, wenn sie von einem Eingang in einen Ausgang führt. Gültig ist sie hingegen, wenn sie von einem Ausgang in einen Eingang führt. Des Weiteren ist eine Verbindung ungültig, wenn die Datentypen des Ausgangs, an dem die Verbindung beginnt, mit den Datentypen des Eingangs, in welchem die Verbindung endet, nicht kompatibel sind. D.h., dass beim Erstellen einer Verbindung sichergestellt werden muss, dass alle Marken, welche der Ausgang weitergibt, vom Eingang und dessen Operationsknoten verarbeitet werden können.

Der Editor muss dem Benutzer Informationen über den Status eines Operationsknotens zur Verfügung stellen. Ein Operationsknoten muss die Möglichkeit haben, während der Ausführung einen Fortschritt anzuzeigen. Und er muss die Möglichkeit haben, seine grafische Ausgabe auf der Arbeitsfläche zu beeinflussen oder selbst eine grafische Ausgabe zu Erzeugen, welche vom Editor an geeigneter Stelle auf der Arbeitsfläche angezeigt wird.

Der Editor muss eine Schnittstelle anbieten, welche Operationen verwenden können, um einen spezifischen visuellen Ausgabebereich auf dem Bildschirm anzulegen und aufzulösen. Dieser Ausgabebereich soll es Knoten ermöglichen, z.B. Texte, Bilder, Tabellen oder andere visuelle Darstellungen direkt für den Benutzer auszugeben. Damit ist nicht die grafische Ausgabe des Operationsknotens auf der Arbeitsfläche gemeint, sondern eine Benutzerschnittstelle außerhalb der Arbeitsfläche.

Der Benutzer muss die Struktur des Programms während seiner Ausführung bearbeiten können.

3.5.3 Testumgebung

Um die Stabilität von Knotentypen zu testen, ist das Beurteilen von verschiedenen Kriterien möglich. Ein Operationsknoten verhält sich stabil, wenn er instanziiert, initialisiert und ausgeführt werden kann, ohne dass eine Ausnahme (*Exception*) auftritt bzw. ein Abbruch des Ausführungspfades der Laufzeitumgebung eintritt. Mit Stabilität in diesem Kontext ist nicht die Korrektheit der Operation gemeint. Eine Überprüfung der Stabilität eines Knotentyps ist also keine Verifikation seiner Operation. Ein Operationsknoten sollte Metadaten besitzen, welche die Interaktion mit einem menschlichen Benutzer komfortabler gestalten (vergl. Ende von 3.2 - Metadaten).

Die Testumgebung von DynamicNodes muss Testfälle für Knotentypen ausführen können, welche die Stabilität überprüfen. Dabei muss ein Testfall auf alle Knotentypen anwendbar sein und eine sinnvolle Aussage liefern oder anders ausgedrückt, darf ein Testfall keine operationsspezifischen Tests durchführen. Wenn für einen Knotentyp alle integrierten Testfälle erfolgreich waren, ist damit eine hohe Wahrscheinlichkeit nachgewiesen, mit der Operationsknoten dieses Knotentyps in Programmen ausgeführt werden können, ohne die Laufzeitumgebung oder andere Knoten in ihrer Arbeit zu stören.

Die Existenz und syntaktische Korrektheit von Metadaten müssen von der Testumgebung überprüfbar sein.

Die Testumgebung muss so entworfen werden, dass nachträglich weitere Testfälle hinzugefügt werden können. Dazu könnten Testfälle zum Speicherplatzbedarf oder zur Kompatibilität mit Erweiterungen zählen.

Um einen Knotentypen zu testen, werden von der Testumgebung Testfälle auf ihn angewendet. Ein Testfall ist eine ausführbare Einheit, welcher ein Knotentyp oder eine Knoteninstanz als Parameter übergeben werden. Nach der Ausführung eines auf *ein* Kriterium des Knotentyps oder der Knoteninstanz beschränkten Tests, liefert der Testfall zurück, ob und wie der Test bestanden wurde.

Es werden zwei Arten von Testfällen unterschieden:

- Klassentests
  Klassentests überprüfen Metadaten, Initialisierungsverhalten und andere Eigenschaften von Knotentypen, welche ohne eine existierende Instanz der Knotentypen überprüfbar sind.

- Instanztests
  Instanztests überprüfen die Ausführbarkeit von Instanzen von Knotentypen innerhalb einer Laufzeitumgebung und andere Eigenschaften, welche sich mit einer Instanz eines Knotentyps überprüfen lassen.

Bei einem Testdurchlauf für einen Knotentypen werden zunächst alle Klassentests auf den Knotentyp angewendet und wenn keiner von diesen einen Fehler gemeldet hat, werden alle Instanztests auf Instanzen dieses Knotentyps angewendet.

Da Knotenbibliotheken von projektfremden Programmierern implementiert werden sollen, muss ein möglichst geringes Vorwissen zur Entwicklung von Knotentypen ausreichen. Die Testumgebung soll ein Werkzeug sein, mit dem überprüft werden kann, ob Knotentypen in einem Programm verwendet werden können ohne den Absturz der Laufzeitumgebung zu verursachen.

Nach dem Prinzip „Je mehr Testfälle, desto stabiler die Knotentypen“ sollte die Testumgebung möglichst viele Testfälle enthalten. Es ist zu erwägen, ob eine Plug-In-Schnittstelle für Testfälle sinnvoll ist, damit neue Testfälle unproblematisch hinzugefügt werden können.

3.5.4 Operationsknotenbibliotheken

Eine Operationsknotenbibliothek (Knotenbibliothek) ist eine Sammlung von Knotentypen. Damit eine Operationsknotenbibliothek von der Laufzeitumgebung geladen werden kann, muss sie eine von DynamicNodes vorgeschriebene Schnittstelle besitzen. Diese Schnittstelle muss es zulassen, dass Operationsknotenbibliotheken in mehreren objektorientierten Sprachen entwickelt werden können.

Eine Operationsknotenbibliothek besteht sowohl aus der Definition von einem oder mehreren Knotentypen als auch den dazugehörigen Metadaten. Unter der Definition eines Knotentyps ist der Quellcode bei interpretierten Sprachen und der Maschinencode bei kompilierten Sprachen zu verstehen.

3.5.5 Hilfesystem

Das Hilfesystem soll dem Benutzer von DynamicNodes Informationen über die Arbeitsweise der Laufzeitumgebung und über die Nutzung der Benutzeroberfläche liefern. Es muss die Integration von Texten und Bildern zur Beschreibung der Funktionalität von Knotenbibliotheken unterstützen. Die Beschreibung der Funktionalität und der Anschlüsse von Knotentypen in einer Knotenbibliothek muss in einem Format vorliegen, welches folgende Eigenschaft besitzt:

- Einfacher Zugriff und Extraktion von einzelnen Informationen von z.B. einem Anschluss eines Knotentyps
- Nachträgliche Anpassung der visuellen Darstellung zur gestalterischen Integration der Knotenreferenz in das Hilfesystem

Das Hilfesystem muss eine Schnittstelle anbieten, über welche die Beschreibung für einen Knotentyp direkt aufgerufen werden kann.

Das Hilfesystem soll in den grafischen Editor derart integriert werden, dass ein gleichzeitiges Anzeigen von Hilfe und grafischer Arbeitsfläche möglich ist.

3.6 Nicht funktionale Anforderungen

Dieser Abschnitt enthält Forderungen für Qualitätsmerkmale nach DIN 66272, bzw. ISO/IEC 9126.

3.6.1 Benutzbarkeit

DynamicNodes soll als Programmierumgebung vielfältige Einsatzmöglichkeiten bieten. Um das zu gewährleisten, werden an die einzelnen Teile des Systems einige weiterführende Anforderungen gestellt.

Gesamtsystem

Die Installation von DynamicNodes auf einem unterstützten Betriebssystem darf keine außergewöhnlichen Vorkenntnisse über DynamicNodes erfordern. DynamicNodes sollte seine Daten derart verwalten, dass es von mehreren Benutzerkonten gestartet werden kann, ohne dass sich die verschiedenen Benutzer stören oder ein Benutzer ohne weiteres auf die Entwicklungen eines anderen Benutzers zugreifen kann.

Laufzeitumgebung

Die Laufzeitumgebung muss als Projektteil so implementiert sein, dass sie sich leicht in andere Projekte einbetten lässt. Das kann sinnvoll sein, wenn in einem Projekt die Leistungsfähigkeit von bereits existierenden Operationsbibliotheken und DynamicNodes-Programmen genutzt werden soll. Um die Einbettung der Laufzeitumgebung in andere Projekte zu erleichtern, sollte sie möglichst wenige externe Abhängigkeiten besitzen.

Benutzeroberfläche

Im Vordergrund der Entwicklung steht eine intuitive Bedienung unter Berücksichtigung der vorherrschenden GUI-Gestaltung von Windows-Anwendungen. Soweit möglich, sollte die Benutzeroberfläche selbsterklärend sein.

Im Hinblick auf den Einsatz von DynamicNodes als Experimentierumgebung im schulischen Bereich sollte das Bedienen der Benutzeroberfläche Spaß machen. Eine ansprechende grafische Gestaltung ist daher ein wichtiges Entwurfsziel.

3.6.2 Zuverlässigkeit

DynamicNodes muss im Rahmen dieser Arbeit nicht die Zuverlässigkeit bieten, die für Produktionsumgebungen gefordert wird. Eine Zuverlässigkeit, welche ein einigermaßen reibungsloses Experimentieren ermöglicht, ist ausreichend. Bei der Beurteilung der Zuverlässigkeit ist besonderer Wert auf die Laufzeitumgebung zu legen.

3.6.3 Effizienz

DynamicNodes muss sowohl mit dem Speicherplatz auf der Festplatte als auch mit den Ressourcen des Arbeitsspeichers sparsam umgehen. Die Laufzeitumgebung ist so zu entwerfen, dass auch große Mengen an Knoten und Graphen ausgeführt werden können. Der Speicherbedarf der Operationsknoten setzt sich aus dem Bedarf der Verwaltungsdaten, welche die Laufzeitumgebung für jeden Knoten benötigt, und dem Bedarf der operationsspezifischen Daten zusammen. Letzterer kann bei der Entwicklung der Laufzeitumgebung nicht beeinflusst werden, sondern obliegt dem Knotenentwickler. Der Speicherbedarf den die Laufzeitumgebung je Knoten benötigt, sollte jedoch so gering wie möglich gehalten werden.

Der Transport von Marken, die Aktivierung von Operationen und die Organisation der Parallelität sind zur Laufzeit die kritischen Punkte hinsichtlich der Ausführungsgeschwindigkeit von Programmen in DynamicNodes. Bei der Implementierung dieser Bereiche ist besonderer Wert auf eine hohe Performance zu legen.

3.6.4 Änderbarkeit

Bei der Architektur des Systems ist Erweiterbarkeit ein wichtiges Entwurfsziel. Neben der Erweiterbarkeit durch die externen Operationsbibliotheken sind weitere Schnittstellen zu schaffen. Bereits bei dem Entwurf der Laufzeitumgebung ist darauf zu achten, dass Graphen später um zusätzliche Fähigkeiten erweitert werden können. Vorstellbar wäre eine dreidimensionale Visualisierung von Programmen, eine Laufzeitanalyse, Filter an Anschlüssen, automatische Verbindungen o.ä.

Sowohl die Datenstruktur der Programme als auch die Persistenz-Mechanismen sind so zu entwerfen, dass zu einem späteren Zeitpunkt eine möglichst große Vielfalt an Erweiterungen ergänzt werden kann.

Die Schnittstellen zwischen den Projektteilen sollten schmal und wohl definiert sein, damit bei der Änderung eines Projektteils möglichst wenige Abhängigkeiten gestört werden.

Der Quellcode von DynamicNodes muss ausreichend kommentiert werden, damit ein projektfremder Programmierer die Möglichkeit hat, sich in vertretbarer Zeit einzuarbeiten.

3.6.5 Übertragbarkeit

Wie bereits in 0 beschrieben, muss die Laufzeitumgebung leicht in andere Projekte eingebettet werden können. Die Laufzeitumgebung und auch der grafische Editor müssen sowohl von fachspezifischen Operationsknoten als auch von konkreten Datentypen für die Markenwerte unabhängig sein.

Durch die Implementierung einer fachspezifischen Operationsbibliothek mit den geeigneten Datentypen muss DynamicNodes in unterschiedlichen Anwendungsgebieten einsetzbar sein. Um die Flexibilität und die Interoperabilität von DynamicNodes, in Hinsicht auf mehrere unterschiedliche Anwendungsgebiete, zu maximieren, sollten möglichst fachübergreifende Datentypen bei der Entwicklung von Operationsknoten zum Einsatz kommen. Dadurch wird die Wiederverwendbarkeit von elementaren Operationen in mehr als einer Domäne möglich und die Komposition von fachübergreifenden Programmen erleichtert.

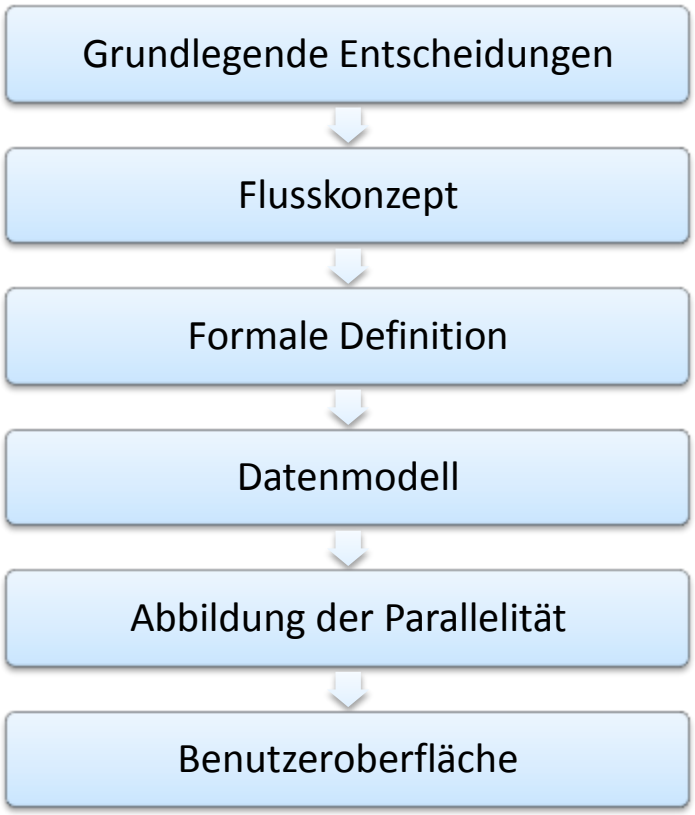
Mindestens die Laufzeitumgebung, besser auch der grafische Editor und die Testumgebung sollten grundsätzlich Plattformunabhängig implementiert werden.

4 Entwurf

Dieses Kapitel beschreibt die Schritte, welche für den Entwurf von DynamicNodes notwendig sind.

Dabei gliedert sich das Kapitel in mehrere Abschnitte. Der erste Abschnitt enthält eine Entscheidung in den vier Grundlegenden Fragen jeder Softwareentwicklung. Der zweite Abschnitt behandelt den Entwurf des Flusskonzepts für DynamicNodes. Der dritte enthält eine formale Definition des Flussmodells auf der Grundlage des zweiten Abschnitts. Im vierten Abschnitt wird das Datenmodell für DynamicNodes-Programme entwickelt. Der fünfte Abschnitt behandelt die Abbildung Parallelität auf das objektorientierte Programmierparadigma und der sechste und letzte Abschnitt enthält den Entwurf der Benutzeroberfläche.

In diesem Kapitel werden häufig kleine Diagramme wie in Abb. 18 verwendet um die Entwurfsentscheidungsmöglichkeiten für einen Entwurfsbereich zu visualisieren. Die Fragen, welche an den meisten Entscheidungsmöglichkeiten stehen, werden nicht immer wortwörtlich oder vollständig im Text aufgegriffen, sondern sollen dem Leser lediglich eine Vorstellung ermöglichen, worum es sich bei dem möglichst kurz gewählten Schlagwort für die jeweilige Entscheidungsmöglichkeit handelt. Oftmals sind solche Fragen weiter oben im Text bereits beantwortet worden und stellen somit eine Beziehung zwischen verschiedenen Entwurfsentscheidungen her.



4.1 Grundlegende Entscheidungen

Bevor der konkrete Entwurf der Programmstruktur und der Laufzeitumgebung beginnen kann, müssen einige grundlegende Entscheidungen getroffen werden. Die vier wichtigsten Entwurfsentscheidungen für ein Software-System sind die Programmiersprache, die Form der Datenhaltung, die Form der Benutzerschnittstelle und die Verteilung des Systems. Unter der Verteilung eines Systems ist zu verstehen, ob und wie eine Software auf mehrere Rechner *verteilt* ausgeführt wird und in welcher Form die verschiedenen Teile der Software über ein Netzwerk zusammenarbeiten.

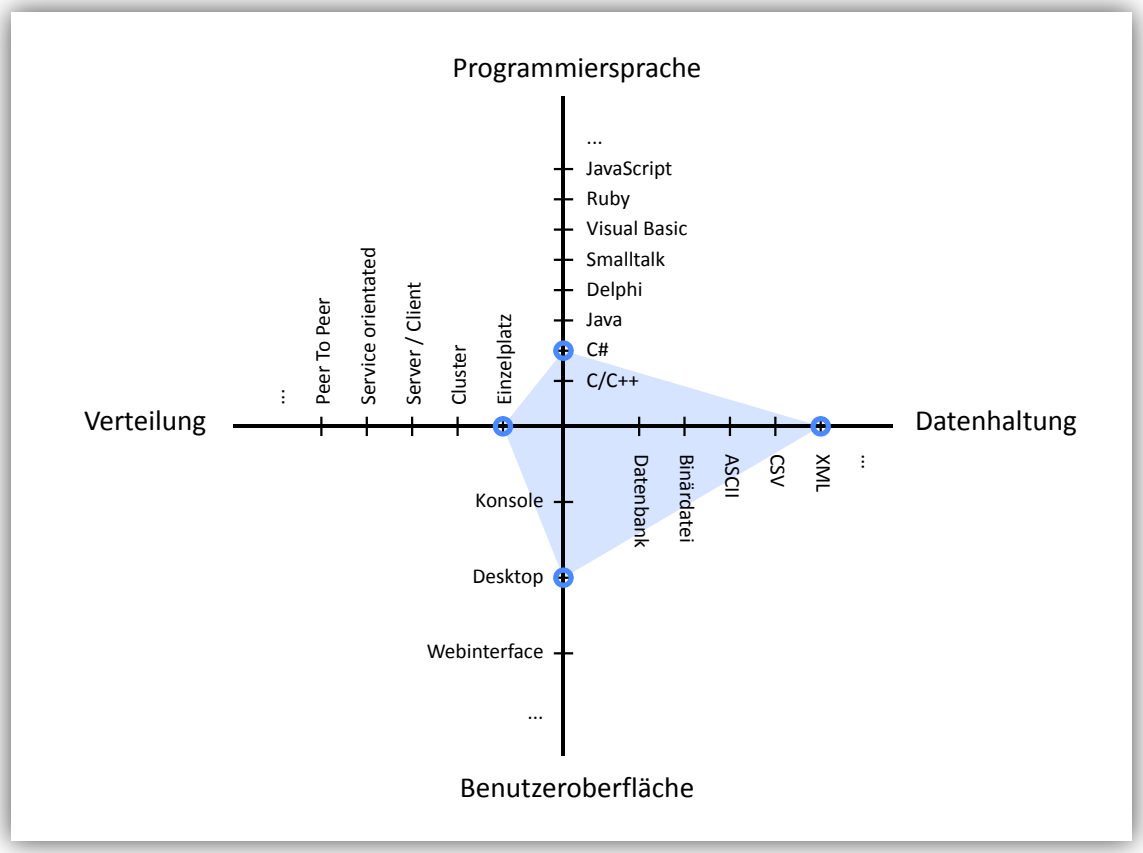


Abb. 15 Grundlegende Entwurfsentscheidungen

4.1.1 Programmiersprache

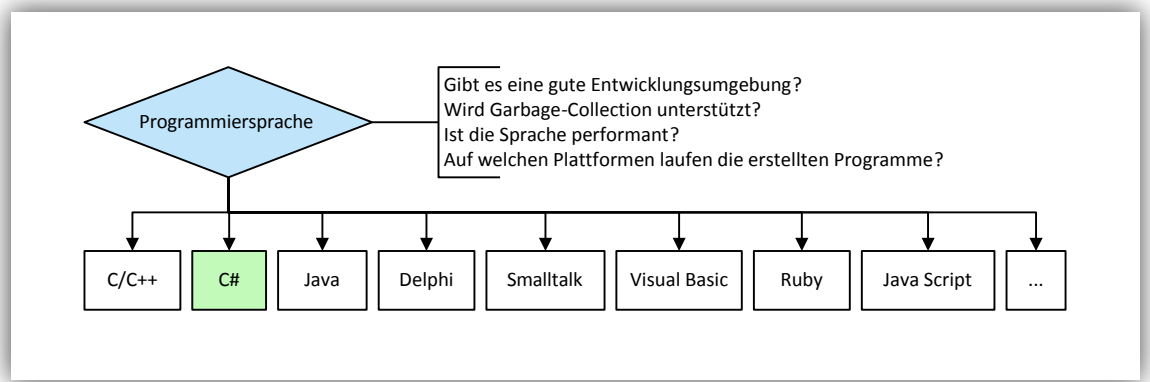


Abb. 16 Entwurfsentscheidungsmöglichkeiten für die Wahl der Programmiersprache

Bevor eine konkrete Programmiersprache ausgewählt wird, sollte geklärt werden, welches Programmierparadigma die Sprache unterstützen soll. Es gibt u.a. imperative, prädikative, deklarative, prozedurale, funktionale, logische, strukturierte und objektorientierte Programmiersprachen.

Diese Begriffe beschreiben Ausrichtungen und Eigenschaften von Programmiersprachen, welche zum Teil auch als Programmierparadigma bezeichnet werden. Auf der Seite 290 von (Ernst, 2000) wird zur Einführung in das objektorientierte Paradigma ein kleiner Überblick gegeben.

„Bei imperativen Programmiersprachen (etwa FORTRAN) wird ein Programm als Folge von einzeln ausführbaren Befehlen aufgefasst, wobei die Ein/Ausgabe über Variablen erfolgt, die zwischengespeichert und verarbeitet werden können. Dieses Prinzip reflektiert stark die Von-Neumann-Architektur.
[…]
Beim objektorientierten Ansatz geht man dagegen von Objekten aus, die nicht nur Daten umfassen, sondern auch eine Beschreibung der damit möglichen Manipulationen, die als Methoden bezeichnet werden. Das erfolgreiche Konzept der strukturierten Programmierung wird damit um das Konzept der objektorientierten Programmierung (OOP) erweitert.“
(Ernst, 2000)

Auch (Balzert, 2005) gibt auf Seite 75 einen Einblick in unterschiedliche grundlegende Konzepte von Programmiersprachen.

Es gibt zwei Gründe, warum hier das objektorientierte Programmierparadigma für die Implementierung von DynamicNodes ausgewählt werden soll:

1. Laut (Balzert, 2005) Seite 78 haben die objektorientierten Sprachen speziell Java, C++ und C# heute die größte Bedeutung. Das ist wichtig, da andere Programmierer ihr erlerntes Wissen nutzen können, um Knotenbibliotheken zu entwickeln.
2. Die Spezifikation schreibt in 3.2 und 3.5.4 explizit vor, dass Knoten mit mehreren objektorientierten Programmiersprachen entwickelt werden sollen. Eine Schnittstelle zwischen Knotenbibliotheken und Laufzeitumgebung ist einfacher zu entwerfen, wenn die Laufzeitumgebung in der gleichen Sprache geschrieben ist, in welcher die Knotenbibliotheken geschrie-

ben werden sollen, bzw. die Sprache für die Laufzeitumgebung das gleiche Programmierparadigma unterstützt, wie die Sprachen für die Knotenbibliotheken.

Kriterien zur Auswahl der Programmiersprache

In der Abb. 17 werden eine Anzahl von objektorientierten Programmiersprachen aufgelistet, welche für eine Auswahl verglichen werden sollen.

Kriterien / Sprache	Kompilierend	Garbage-Collection	Plattformunabhängig	Komfortable Entwicklungsumgebung	Einfacher XML-Zugriff	Oberflächenbibliothek	Binärkompatibel mit anderen Sprachen	Explizites Binden zur Laufzeit	Für .NET / Mono kompilierbar	Programmiererfahrung
C++	●		○	●	●	●	●	○	●[4]	●
D	●	●	○		●	●	●	○		○
Java	●[1]	●	●	●	●	●	●[3]	●	●[5]	●
C#	●[2]	●	●	●	●	●	●	●	●	●
Visual Basic 8	●[2]	●	●	●	●	●	●	●	●	●
Delphi	●		○	●	●	●	●	○	○	
Smalltalk	○	●	●	●	●	●	●[3]	●		
Python		●	●	●	●	●	●[3]	●	●	
Ruby		●	●	●	●	●	○	●		
PHP		●	●	●	●	○	○	●	○	●
Java Script		●	●	○	●			●		●
JScript	●[2]	●	●		●	●	●	●	●	○

Abb. 17 Gegenüberstellung von objektorientierten Programmiersprachen

● Merkmal wird unterstützt

○ Merkmal wird bedingt unterstützt

[1] Java-Byte-Code ist eine Zwischensprache

[2] Die IL für .NET ist eine Zwischensprache

[3] Kann in Java-Byte-Code kompiliert werden

[4] Mit „Managed Extensions for C++“ (Diese Spracherweiterung ermöglicht es, .NET-Programme mit C++ zu implementieren.)

[5] In der Java-Variante J# von Microsoft, welche nicht weiterentwickelt wird, oder mit Hilfe des JIT-Compiler von IKVM.

Zur Gegenüberstellung der in Abb. 17 aufgeführten Programmiersprachen sind die folgenden Kriterien von Bedeutung:

- Kompilierend

  Eine Sprache kann vor der Ausführung in Maschinencode übersetzt (kompiliert) werden oder während der Ausführung direkt aus dem Programmcode interpretiert werden.

- Garbage-Collection
  Einige Sprachen erfordern die manuelle Verwaltung des Arbeitsspeichers. Andere verwenden einen Garbage-Collector um allokierten Speicher automatisch freizugeben, wenn dieser nicht mehr benötigt wird.
- Plattformunabhängigkeit
  Ein mit der Programmiersprache verfasstes Programm sollte ohne Änderungen auf unterschiedlichen Betriebssystemen lauffähig sein. Im Besonderen ist dabei Microsoft Windows und Linux zu berücksichtigen.
- Komfortable Entwicklungsumgebung
  Unter „komfortabel“ wird in diesem Kontext die Unterstützung des Programmierers mit Hilfe von Syntax-Highlighting, Code-Vervollständigung und Refactoring verstanden. Ein wichtiges Merkmal ist zusätzlich ein integrierter Debugger.
- Einfacher XML-Zugriff
  Einige Programmiersprachen bringen eine Programmierbibliothek für einen Zugriff auf XML-Daten mit. Dabei kommen häufig Standards oder Pseudostandards wie DOM oder SAX zum Einsatz.
- Oberflächenbibliotheken
  Für die effiziente Oberflächenprogrammierung ist die Existenz einer Programmierbibliothek nützlich, welche die Elemente der grafischen Schnittstelle (Fenster, Steuerelemente) auf das Programmierparadigma der Wahl abbildet.
- Binäre Kompatibilität mit anderen Sprachen
  Wenn das Kompilat einer Sprache mit dem Kompilat einer anderen Sprache zu einer ausführbaren Einheit verbunden werden kann, sind sie binär kompatibel.
- Explizites Binden zu Laufzeit
  Damit ist die Fähigkeit gemeint, Programmteile, welche zur Entwicklungszeit nicht zur Verfügung standen, zur Laufzeit zu laden und auszuführen. Z.B. kann mit explizitem Binden eine Programmbibliothek geladen werden, deren Name und Klassen/Prozeduren zur Entwurfszeit noch nicht bekannt sind.
- Für Microsoft .NET-Framework oder das Projekt Mono kompilierbar
  Es gibt eine Gruppe von Sprachen, die in die Intermediate Language von Microsoft .NET übersetzt werden können. Diese Sprachen sind automatisch „binär kompatibel“ und, bis auf einige Einschränkungen, auch mit der plattformunabhängigen Laufzeitumgebung Mono lauffähig.
- Programmiererfahrung des Autors
  Es ist wünschenswert, dass der Autor ausreichend Erfahrung in der Verwendung der auszuwählenden Programmiersprache bzw. ihrer Konzepte besitzt, um DynamicNodes in der zur Verfügung stehenden Zeit zu implementieren. Dazu kann, mangels eines adäquaten Maßstabes, keine exakte Aussage gemacht werden. Jedoch sollte eine Programmiersprache gewählt werden, in welcher der Autor mindestens ein kleineres Projekt erfolgreich umgesetzt hat, um den Zeitaufwand zum Erlernen einer neuen Sprache bzw. neuer Konzepte zu vermeiden.

Eingrenzung der Programmiersprachen

Bevor eine endgültige Auswahl getroffen wird, soll zunächst die Menge der zur Auswahl stehenden Sprachen anhand einiger Kriterien dezimiert werden.

Um zu entscheiden ob eher eine kompilierte oder eine interpretierte Sprache in Frage kommt, sollte ein Blick auf die Ausführungsgeschwindigkeit geworfen werden. Die Performance der Laufzeitumgebung wird ein kritischer Faktor im Hinblick auf die Tauglichkeit des Gesamtsystems, denn sie bildet eine zusätzliche Schicht zwischen dem Betriebssystem und den auszuführenden DynamicNodes-Programmen. Im Hinblick auf dieses Kriterium ist eine Sprache, welche in Maschinencode übersetzt werden kann, interpretierten Sprachen vorzuziehen.

Es sollte eine komfortable Entwicklungsumgebung für die Programmiersprache der Wahl verfügbar sein, damit die Implementierungsarbeit von einer einzelnen Person in vertretbarer Zeit bewältigt werden kann.

Es muss möglich sein, mit der Programmiersprache eine Schnittstelle zu beschreiben, welche externe Knotenbibliotheken zur Laufzeit in das DynamicNode-System lädt und zur Ausführung der Flussgraphen verwendet. Auch die Form der angestrebten Benutzerschnittstelle ist bei der Wahl der Programmiersprache von Bedeutung. Die Programmiersprache sollte eine umfangreiche Bibliothek für die Oberflächenprogrammierung mitbringen. Der Zugriff auf XML-Dateien sollte ebenfalls von vorhandenen Bibliotheken unterstützt werden.

Nach den oben formulierten Bedingungen bleiben vier Programmiersprachen übrig, welche der Autor ausreichend beherrscht um die Implementierung bewältigen zu können. Das sind Visual C++ mit MFC, Visual C# und Visual Basic 8 mit dem .NET-Framework und Java 5 mit Swing oder SWT. Visual C# ist die eigentliche .NET-Sprache und ermöglicht gegenüber Visual Basic 8 das Schreiben von kompakterem Code, da sich die Syntax an C anlehnt und nicht auf der ausladende Syntax von BASIC basiert. Wenn also eine .NET-Sprache in Frage kommt, ist C# die bessere Wahl. Um die Auswahl weiter einzugrenzen, ist die Entscheidung zwischen C++ und den beiden Sprachen Java und Visual C# sinnvoll. Denn zwischen diesen beiden Gruppen gibt es wesentliche Unterschiede, welche als Entscheidungshilfe dienen können.

C++ vs. Java und C#

C++ wird mittels Compiler direkt in Maschinencode übersetzt und erfordert eine manuelle Speicherverwaltung. Die beiden anderen Sprachen verwenden im Gegensatz dazu eine Art Zwischencode (Java-Byte-Code, Intermediate Language) und eine Laufzeitumgebung (Java-VM, Common Language Runtime) mit Garbage-Collection.

Das am häufigsten angeführte Argument für C++ ist in diesem Kontext, dass der echte Maschinencode, welcher vom Compiler produziert wird, ein schnelleres Laufzeitverhalten hat, weil der zusätzliche Übersetzungsschritt von einem Zwischencode in echten Maschinencode zur Laufzeit entfällt. Desweiteren wird häufig argumentiert, dass die manuelle Speicherverwaltung in C++ effizienter und perfor-

manter sei, als die Garbage-Collection von Java oder .NET. Beide Argumente mögen in einem direkten Vergleich unter Umständen zutreffen. Jedoch spricht auch eine Menge für die anderen Sprachen.

Der Zwischencode, welcher aus Java und C# erzeugt wird, ist grundsätzlich plattformunabhängig. D.h., dass kompilierte Java- oder C#-Programme auf unterschiedlichen Betriebssystemen laufen können. Die Verfügbarkeit der Laufzeitumgebung für unterschiedliche Betriebssysteme ist jedoch für die tatsächliche Plattformunabhängigkeit ausschlaggebend. Die Laufzeitumgebung für Java 5 existiert für die meisten populären Betriebssysteme wie Windows, Linux, Unix, Solaris oder Mac OS X. Das .NET-Framework von Microsoft existiert zunächst nur als Implementierung für Windows. Unter dem Namen Mono wird von Novell eine .NET-kompatible Open-Source-Implementierung gepflegt, welche ebenfalls unter Windows, Linux, Unix, Solaris und Mac OS X lauffähig ist.

Der Garbage-Collector ist für die einen ein Stein des Anstoßes, für die anderen ein Segen. Die Entwicklungszeit und Fehleranfälligkeit von Programmen, welche einen Garbage-Collector nutzen, ist erwiesener Maßen geringer. In dem Siegeszug der aufkommenden Sprache D wird sogar behauptet, dass Programme mit Garbage-Collection im Gesamtbild schneller laufen, weil der Garbage-Collector die Wartezeiten des Systems nutzt, um ungenutzten Speicher freizugeben, wohingegen bei manueller Speicherverwaltung meist die wertvolle Zeit innerhalb der Abarbeitung von rechenintensiven Arbeitsschritten zusätzlich für die Speicherverwaltung benutzt wird (Digital Mars). Das direkte kompilieren in Maschinencode spricht zunächst für C++, die zwingende Plattformabhängigkeit von kompilierten Programmen von Visual C++ aber dagegen. Die Garbage-Collection sollte im Rahmen dieses Vergleichs weniger aus der Sicht der Ausführungsgeschwindigkeit, als vielmehr aus der Sicht des Komforts während der Programmierung betrachtet werden.

Die nun diskutierten Merkmale der Sprachen sind jedoch nicht die allein ausschlaggebenden Kriterien für eine endgültige Entscheidung. Es gibt eine bemerkenswerte Fähigkeit die sowohl Java 5 als auch die .NET-Sprachen mitbringen. Diese *Reflection* genannte Fähigkeit wird durch Meta-Daten im Zwischencode möglich und erlaubt es, das Programm selbst zum Gegenstand der Verarbeitung zu machen. Das heißt z.B., dass zur Laufzeit für ein Objekt die Member bestimmt und aufgerufen werden können, obwohl deren Namen zur Entwurfszeit unbekannt sind. Oder dass eine Klasse als Parameter übergeben werden kann.

Reflection ist auch wichtig im Kontext des expliziten Bindens. Dabei ist es möglich, Programmbibliotheken zur Laufzeit zu laden und nach Klassen und deren Methoden zu durchsuchen, sowie diese zu instanziieren bzw. aufzurufen. Diese Fähigkeit ist deshalb so wichtig für DynamicNodes, weil ein wesentliches Merkmal der Laufzeitumgebung darin besteht, externe Knotenbibliotheken zur Laufzeit einzubinden. *Reflection* macht das Entwerfen einer entsprechenden PlugIn-Schnittstelle erheblich einfacher.

Und nicht zuletzt ist es auch die Erfahrung in einer Sprache, welche einen Entwickler in seiner Arbeit effizient oder ineffizient macht. Der Autor hatte während seines Studiums die Gelegenheit, alle drei Sprachen kennenzulernen und kleinere Projekte damit umzusetzen.

Sowohl der Garbage-Collector als auch die *Reflection*-API sprechen im Rahmen dieser Arbeit gegen C++. Übrig bleiben die beiden Sprachen Visual C# und Java 5, zwischen denen ein direkter Vergleich angebracht ist um eine endgültige Wahl zu treffen.

Java vs. C#

Beide Sprachen bringen eine komfortable und kostenlose Entwicklungsumgebung mit. Für C# mit .NET 2.0 ist das Microsoft Visual C# Express 2005 und für Java 5 das Eclipse 3.3 kostenlos verfügbar. Die Sprachen sind sich sehr ähnlich. Beide besitzen Namensräume bzw. Pakete, explizite Schnittstellen, eine *Reflection*-API, Attribute bzw. Annotations, Generics und umfangreiche Bibliotheken für die Oberflächenprogrammierung. Auch APIs für die XML-Programmierung sind für beide Sprachen verfügbar.

Das einzige Kriterium, welches nun endgültig zu Microsoft .NET bzw. Mono und C# führt, ist die Tatsache, dass Microsoft das .NET-Framework von Anfang an für die Verwendung mit mehreren Sprachen ausgelegt hat. So ist es möglich eine Klassenbibliothek mit C#, eine andere mit Visual Basic 8 und die dritte mit C++ und Managed Extensions zu entwickeln. Auch andere Sprachen wie JScript oder Delphi können in .NET-Programmcode übersetzt werden. Laut der Spezifikation in 3.5.4 muss es möglich sein, Knotenbibliotheken in mehr als einer objektorientierten Sprache zu entwickeln. Wenn nun DynamicNodes selbst in Visual C# entwickelt wird, ist es kein Problem Knotenbibliotheken in C#, Visual Basic oder JScript zu implementieren. Das Ergebnis ist in jedem Fall ein .NET-Assembly in der Intermediate Language, welches zur Laufzeit geladen, per *Reflection* analysiert und ausgeführt werden kann.

4.1.2 Datenhaltung

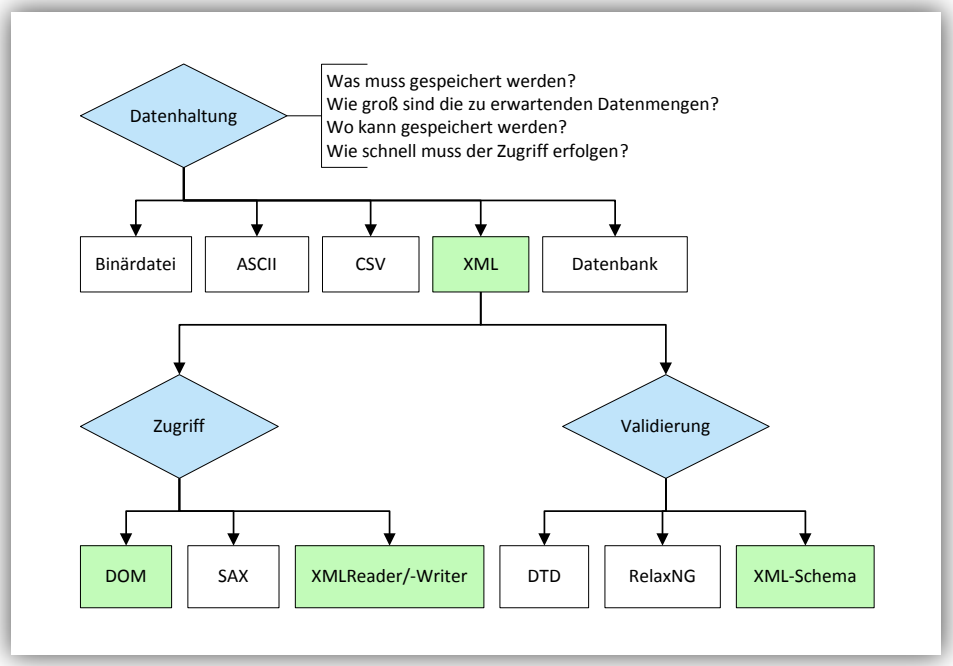


Abb. 18 Entwurfsentscheidungsmöglichkeiten für die Datenhaltung

Bei dem Entwurf von DynamicNodes ist zu entscheiden, welche Daten wie und wo gespeichert werden sollen.

Art und Größenordnung der Daten

Zunächst ist zu klären welche Daten verwaltet bzw. gespeichert werden sollen. Für das DynamicNode-System selber sind dies lediglich einige Programmeinstellungen. Und natürlich sollen die Flussgraphen, welche in DynamicNodes entwickelt wurden, gespeichert und geladen werden können. Dabei handelt es sich im ersten Fall lediglich um eine Menge von benannten Eigenschaften, welche als Zeichenkette darstellbar sind. Die Flussgraphen-Programme selber bestehen aus einer strukturierten Menge von Knoten, Anschlüssen und Verbindungen. Wobei die Datenmengen jedoch verhältnismäßig gering ausfallen.

Im einfachsten Fall reicht für die Beschreibung eines Knotens in einem Flussgraphen der Klassenname der Knotenklasse bzw. die ID des Knotentyps aus. Anschlüsse sind durch den Knoten selbst definiert. Die Verbindungen können als eingehende Verbindungen für jeden Knoten gespeichert werden. Dabei reicht eine Identifikation des verbundenen Knotens, des Ausgangs und des Eingangs. Wenn nun angenommen wird, dass ein von einem Benutzer erstelltes Netz weniger als 100 Knoten enthält und je Knoten durchschnittlich 500Bytes für die Identifikation und die Beschreibung der Verbindungen notwendig ist, fallen für ein Netz weniger als 50KB an Daten an. Welche Datenmengen durch die in der Spezifikation geforderten Erweiterungen von Graph, Knoten und Anschlüssen anfallen, ist nicht abzuschätzen.

Speicherort

Zunächst sollte eingegrenzt werden, an welchem Ort die Daten gespeichert werden sollen. Typischerweise arbeitet ein Benutzer an einem Flussgraph und möchte ihn später ausführen oder ändern. Dazu ist es sinnvoll den Flussgraph auf der Festplatte des Arbeits-PCs des Benutzers zu speichern. In industriellem Umfeld wäre es denkbar, dass Flussgraphen in einer zentralen Datenbank gespeichert werden sollten, jedoch wird dieses Umfeld bei der konkreten Entwicklung der Datenhaltung zunächst nicht berücksichtigt. Vielmehr soll im DynamicNode-System eine Schnittstelle für die Datenhaltung definiert werden, mit welcher die Umstellung auf eine andere Form der Datenhaltung keine Auswirkung auf die Kontenbibliotheken und geringe Auswirkungen auf die Laufzeitumgebung hat.

Dateiorganisation

Es gibt zwei grundlegende Arten eine Datei zu organisieren. Die Erste ist ein System-eigenes proprietäres Binärformat, mit dem die Daten in kompakter Form und beliebig strukturiert gespeichert werden können. Die Zweite ist eine textuelle Repräsentation der Daten. Diese textuelle Repräsentation kann ebenfalls in einem proprietären Format mit ASCII-Codes geschehen oder es wird ein Standard zur Formatierung verwendet. Dabei kommt z.B. CSV für tabellarische Daten in Frage oder eine Meta-

Sprache wie XML. Mit XML ist die Definition eines eigenen angepassten Formates innerhalb eines verbreiteten Standards möglich.

Für die Binärform sprechen ein effizienter Zugriff und die Vermeidung von Overhead, wodurch Speicherplatz gespart werden kann. Nachteilig ist die Notwendigkeit den Zugriff auf die Dateien im Detail implementieren zu müssen. Eine proprietäre textuelle Repräsentation bringt denselben Implementierungsaufwand mit sich und benötigt zusätzlich mehr Speicherplatz. Die Nutzung von Standards wie CSV ist nur sinnvoll, wenn die Daten eine tabellarische Struktur besitzen. Die Vorteile eines Meta-Formates wie XML sind die Verfügbarkeit von Programmierbibliotheken, welche den Zugriff erleichtern, eine große Anzahl von Werkzeugen zur Verarbeitung von derart formatierten Daten und im Falle von XML eine Möglichkeit, die syntaktische Korrektheit von Dateien zu überprüfen. Als Nachteil ist sicherlich der Overhead bei XML zu betrachten, welcher jedoch bei den oben abgeschätzten Größenordnungen in Kauf genommen werden kann.

Auch die Möglichkeit der Inspektion durch einen Benutzer spricht für XML. Bei Binärdateien ist es als Benutzer oder Entwickler nahezu unmöglich, bei ASCII-Dateien recht umständlich, den Inhalt der Dateien zu erfassen. Bei XML hingegen ist dies, bei entsprechender Nutzung von Zeilenumbruch und Einrückung, auch bei großen Dateien möglich.

Zum jetzigen Entwicklungsstand ist noch nicht genau abzusehen, welche Struktur die Graphen bei der Speicherung besitzen. Jedoch ist abzusehen, dass eine Repräsentation in XML möglich ist, da in XML sowohl hierarchische als auch listenartige Strukturen abgebildet werden können. Somit ist XML die erste Wahl für die Speicherung der DynamicNodes-Graphen.

Zugriff auf XML-Daten

Für das Lesen und Schreiben von XML gibt es eine Reihe von standardisierten Vorgehensweisen. Eine Möglichkeit bietet der Standard DOM[2]. Bei dieser Methode wird das XML-Dokument komplett eingelesen und in einer Baumstruktur im Arbeitsspeicher gehalten. Anschließend ist es möglich, mit einer standardisierten Anzahl von Zugriffsfunktionen die Elemente des Dokumentenbaums zu lesen, zu ändern, neue Elemente einzufügen oder Elemente aus dem Baum zu löschen. Wenn die Arbeit an dem Dokument abgeschlossen ist, kann es wieder als XML-Dokument in einer Datei abgelegt werden.

Um Elemente in einem Dokument zu adressieren, gibt es die Sprache XPath, mit dessen Ausdrücken die Elemente des Dokumentenbaumes nach vielfältigen Kriterien gefiltert werden können. Der DOM-Standard hat den Vorteil, dass es möglich ist, wahlfrei Elemente des XML-Dokumentes zu adressieren, zu lesen und zu manipulieren. Der Nachteil von DOM ist der Aufwand das gesamte Dokument im Arbeitsspeicher zu halten. Einen DOM-Baum könnte man als *random access medium* bezeichnen. Dieses Vorgehen ist nur bei kleinen und mittelgroßen Dokumenten sinnvoll.

[2] Document Object Model

Eine andere Vorgehensweise ist die SAX[3]. Die SAX basiert auf einem Ereignis-Modell (*event driven*) und einem rein sequentiellen Lesen der XML-Daten. Der Benutzer registriert eine Reihe von Callback-Funktionen und startet das Lesen des Dokumentes. Während dem Durchlaufen des Dokuments werden für die einzelnen Elemente die jeweiligen Callback-Funktionen aufgerufen. Mit SAX lässt sich eine Art XML-Fluss-Architektur umsetzen. Dabei bietet jeder SAX-Ereignis-Empfänger wiederum die SAX-Schnittstelle an. Dadurch ist der Aufbau einer Art Filterkette möglich. Der Vorteil von SAX ist, dass nahezu kein Arbeitsspeicher benötigt wird und die Verarbeitung sofort nach dem Öffnen des Dokumentes beginnen kann. Die Verarbeitung selbst ist sehr schnell. Im Vergleich zu DOM entfällt das Aufwändige Laden aller Daten in den Arbeitsspeicher. Der Nachteil ist, dass kein wahlfreier Zugriff möglich ist.

Ein dritter Ansatz ist, im Gegensatz zu dem Ereignis-Modell von SAX, ein Nachfrage-orientiertes Modell (*demand driven*). Dieses Modell wird z.B. von dem XML-Reader im .NET-Framework (`System.Xml.XmlReader`) umgesetzt. Bei diesem Ansatz ist ähnlich dem SAX nur ein sequentielles Lesen möglich. Jedoch fragt der Benutzer des XML-Readers ausdrücklich nach einem bestimmten Element des XML-Dokumentes und der Reader durchläuft das Dokument bis er auf das beschriebene Element stößt. Wurde es gefunden, kann der Benutzer beispielsweise die Attribute und den Inhalt des Elementes erfragen. Anschließend wird die nächste Anfrage gestellt, welche den Reader weiter durch das Dokument laufen lässt. Ein Rücksprung ist jedoch nicht möglich. In Arbeitsspeicheraufwand und Verarbeitungsgeschwindigkeit ähnelt dieses Konzept der SAX. Durch die umgekehrte Herangehensweise ist jedoch mal das Ereignis-Modell und mal das Nachfrage-Modell sinnvoller.

Im .NET-Framework gibt es den XML-Writer (`System.Xml.Xmlwriter`) als ein Pendant zum XML-Reader, welcher das formatierte Schreiben von XML-Dokumenten ermöglicht. Generell ist jedoch das Schreiben von XML-Dokumenten aufgrund der einfachen Syntax-Regeln einfacher als das Lesen.

Für das Lesen von DynamicNodes-Flussgraphen ist wahrscheinlich DOM eine gute Wahl, da die Größe der Dokumente begrenzt ist und noch nicht abzusehen ist, ob ein rein sequentieller Zugriff für die effiziente Rekonstruktion eines Graphen aus der XML-Datei ausreicht. Für das Lesen von XML-Konfigurationsdateien ist die SAX wohl genauso geeignet wie der XML-Reader. Da der XML-Reader jedoch bereits im .NET-Framework enthalten ist, ist dieser vorzuziehen.

Validierung von XML-Daten

Um die Syntax von XML-Dateien zu validieren, gibt es mehrere Möglichkeiten. Es haben sich in den letzten Jahren unterschiedlich komplexe Sprachen zur Beschreibung von XML-Schemata entwickelt. Ich möchte nur einige davon nennen. Die klassische Variante ist die Document Type Definition kurz DTD, welche jedoch selbst nicht auf XML beruht und deshalb für neue Projekte nicht unbedingt zu empfehlen ist. RelaxNG ist eine Entwicklung, die versucht die Syntax der Schema-Definitionen so kurz wie möglich zu halten. Dadurch haben RelaxNG-Schemata den Ruf für den Menschen besser lesbar

[3] Simple API for XML

zu sein. RelaxNG basiert selbst auf XML, bietet jedoch zusätzlich eine noch kompaktere nicht XML-Syntax, welche für die manuelle Entwicklung von Schemata beliebt ist.

XML-Schema ist, neben der DTD, die wohl am meisten verbreitete Sprache zur Validierung von XML-Daten. XML-Schema basiert selbst auf XML und besitzt die Möglichkeit Datentypen mit Hilfe von Vererbung zu definieren. Auch deshalb ist diese Sprache recht komplex. Aufgrund der guten Unterstützung von XML-Schema in Visual Studio und Microsoft .NET ist die Komplexität der Sprache jedoch leicht zu handhaben, deshalb soll für die Definition der XML-Schemata in DynamicNodes XML-Schema verwendet werden.

4.1.3 Benutzeroberfläche

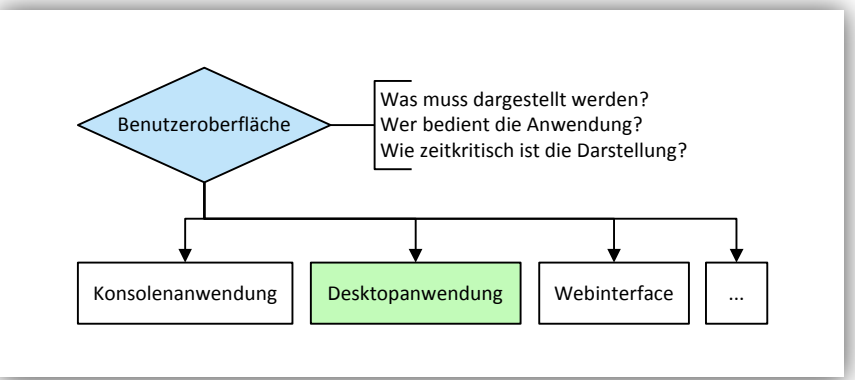


Abb. 19 Entwurfsentscheidungsmöglichkeiten der Benutzeroberfläche

Die dritte grundlegende Entscheidung für einen Softwareentwurf ist die Form der Benutzeroberfläche. Es gibt generell verschiedene Formen von Benutzeroberflächen. In Frage kommt evtl. eine Kommandozeilen-basierte Benutzerschnittstelle, ein Webinterface oder eine klassische Desktopanwendung.

Die Spezifikation fordert einen grafischen Editor für den Entwurf von Flussgraphen. Durch diese Forderung scheidet die Kommandozeile für weitere Untersuchungen bereits aus. Ein Webinterface mit Drag&Drop-Fähigkeiten, mehreren Unterfenstern und einer generell performanten Maus-Ereignis-Verarbeitung ist zwar mit AJAX[4] vorstellbar, aber für DynamicNodes als Arbeitsplatz-orientierte Anwendung wenig sinnvoll. Die Anwendung wird sowohl von ausgebildeten Informatikern als auch von ungelernten Anwendern bedient werden und muss zum Experimentieren geeignet sein. Damit bleibt eine klassische Desktopoberfläche übrig, denn diese kann mühelos die genannten Anforderungen erfüllen.

Ausgehend von der Wahl der Programmiersprache C# und dem .NET-Framework als Basis-Bibliothek liegt es nahe, deren Unterstützung für Benutzeroberflächenprogrammierung zu untersuchen. Mit Windows-Forms[5] ist im .NET-Framework bereits eine leistungsfähige Bibliothek für Fenster-

[4] Asynchronous JavaScript and XML - eine Schlüsseltechnologie bei der Erstellung interaktiver Webseiten
[5] Programmierschnittstelle bzw. Klassenbibliothek für Grafik- und Oberflächenprogrammierung in .NET

orientierte Anwendungen enthalten. Windows-Forms bietet ein objektorientiertes Klassenmodell für die Programmierung von Fenstern und Steuerelementen. Eine Vielzahl von standardisierten Steuerelementen ist in Windows-Forms bereits enthalten, u.a. auch sehr komfortable und anpassungsfähige Exemplare wie eine tabellarische Datenansicht (`DataGridView`) oder eine generische Eigen-Eigenschaftsliste (`PropertyGrid`). Neue Steuerelemente können leicht aus bestehenden komponiert oder vollkommen neu programmiert werden.

4.1.4 Verteilung

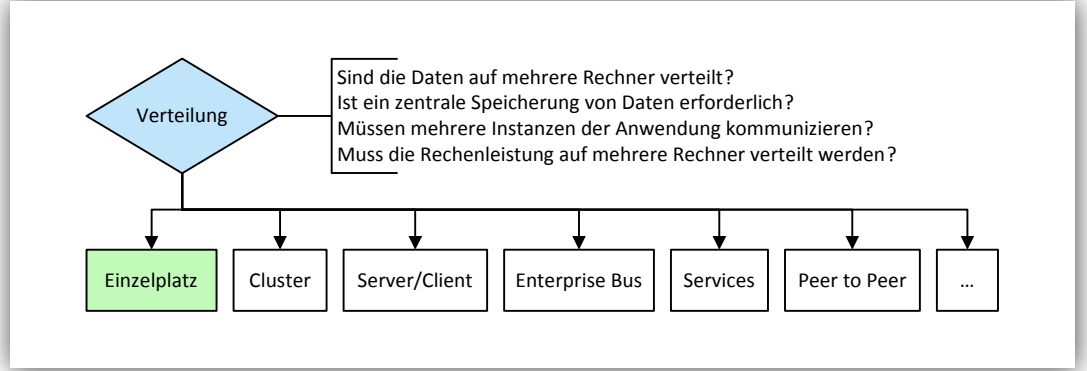


Abb. 20 Entwurfsentscheidungsmöglichkeiten der Verteilung

Die letzte grundlegende Entwurfsentscheidung ist die Frage der Verteilung. In der heutigen Softwarelandschaft sind verteilte Anwendungen durch das allgegenwärtige Internet weit verbreitet. Für die verschiedensten Anwendungskonzepte wird die Möglichkeit genutzt Daten und Rechenleistung auf mehrere Rechnern zu verteilen. Nun ist zu klären, ob dies auch für ein Programmiersystem wie DynamicNodes sinnvoll ist.

Der Benutzer entwirft die Flussgraphen in einer Desktopanwendung und führt sie auf demselben PC aus, auf dem er sie entwirft. Die Flussgraphen-Programme sollen auf der Festplatte eines PCs abgelegt werden. Laut der Spezifikation ist die Nutzung von mehreren Prozessoren (Prozessorkernen) bei der Ausführung eines Flussgraphen gefordert, nicht jedoch die parallele Ausführung auf mehreren Rechnern. Infolge dessen ist eine Verteilung von DynamicNodes selbst nicht notwendig.

Ob einzelne Aspekte von Verteilung in der Anwendung nicht dennoch zum Tragen kommen, ist jedoch noch unklar. Denn es ist durchaus denkbar, eine Knotenreferenz im Sinne eines Hilfesystems auf einem Webserver abzulegen oder etwa Knoten zu implementieren, welche Daten zwischen Rechnern übertragen können. Diese Möglichkeiten sind für den Entwurf des Programmiersystems jedoch zunächst unerheblich.

4.2 Flusskonzept

In diesem Abschnitt wird der Entwurf des Flusskonzepts für DynamicNodes-Programme dokumentiert.

Datenfluss-Systeme können nach verschiedenen Kriterien unterschieden werden (vergl. 2.1.2). Diese Kriterien sollen an dieser Stelle dazu dienen, den Entwurfsraum für das Flusskonzept aufzuspannen. Dabei sollen folgenden Dimensionen berücksichtigt werden:

- Aktivierungsprinzip
- Vorbedingung
- Nachbedingung
- Knotenerzeugung
- Verbindungskapazität
- Unterscheidung von parallelen Marken

Unter der *Vorbedingung* ist sowohl die Voraussetzung als auch der Auslöser für die Aktvierung von Knoten zu verstehen. Die *Nachbedingung* beschreibt was nach einer Knotenaktivierung zwingend geschehen sein muss. Hinter dem Punkt *Knotenerzeugung* verbirgt sich die Art und Weise wie der Programmcode und die Daten von Knoten und Knotentypen verwaltet werden, wann Instanzen von Knoten erzeugt werden und wie der Fluss im Graph bei Schleifen organisiert wird. Die *Verbindungskapazität* beschreibt, wie viele Marken eine Verbindung tragen kann. Und die *Unterscheidung von Parallelen Marken* beschreibt, wie Marken, welche durch eine mehrfache (parallele) Ausführung eines Knotens erzeugt wurden (z.B. bei einer Schleife), unterschieden werden.

4.2.1 Aktivierungsprinzip

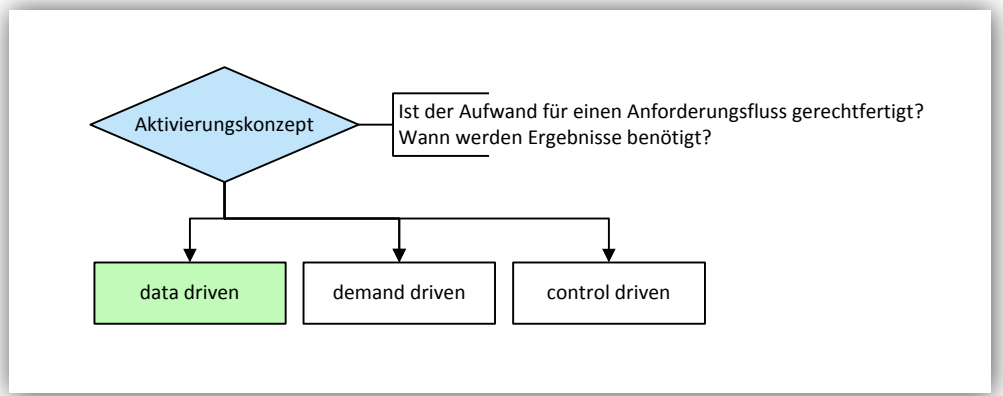


Abb. 21 Entwurfsentscheidungsmöglichkeiten für das Aktivierungsprinzip der Knoten

Zunächst sollte geklärt werden, ob das Flusskonzept *data driven*, *demand driven* oder *control driven* arbeiten soll. Eine Steuerung durch einen Kontrollfluss hebt die Vorteile der dezentralen Flussgraphen-Architektur auf und wird an dieser Stelle nicht in Betracht gezogen.

Wenn ein Flusssystem *data driven* arbeitet, wird ein Knoten aktiviert, wenn ausreichend Marken auf seinen eingehenden Kanten liegen. Wenn ein Knoten aktiviert wird, kann er seine Operation mind. einmal ausführen (vergl. 2.1.2). Zum Abschluss seiner Operation legt der Knoten die Ergebnismarken auf seine ausgehenden Verbindungen. Damit ist unter Umständen die Aktivierung eines nachfolgenden Knoten möglich usw. Die Marken bilden sozusagen den Auslöser für die Aktivierung der Knoten

und setzen sich wie eine Welle im Graph fort, bis alle Marken von Senken konsumiert wurden. Bei diesem Ansatz werden *alle* Knoten ausgeführt, denen ausreichend Marken zur Verfügung stehen.

Ein System welches *demand driven* arbeitet, funktioniert genau umgekehrt. Es wird das Ergebnis eines Knotens gefordert. Falls dem Knoten an den eingehenden Kanten nicht genügend Marken zur Verfügung stehen fordert dieser von seinen vorhergehenden Knoten jene Ergebnismarken, die ihm fehlen usw. Diese Anforderungswelle setzt sich im Graph fort, bis für alle Knoten, welche zur Berechnung des ursprünglich geforderten Ergebnisses *nötig* sind, genügend Marken zur Verfügung stehen. Nach der Anforderungswelle, welche sich gegen die Flussrichtung im Graph fortsetzt, läuft nun eine Welle mit den Ergebnismarken in Flussrichtung bis zu dem Knoten, dessen Ergebnis gefordert wurde (vergl. 2.1.2). Bei diesem Vorgehen ist eine Anforderung der Auslöser für die Aktivierung von Knoten.

Der Vorteil der zweiten Vorgehensweise ist, dass nur jene Knoten aktiviert werden, welche zur Berechnung des benötigten Endergebnisses erforderlich sind. Ein Nachteil ist der zusätzliche Aufwand für den Anforderungsfluss.

Um zwischen den beiden Konzepten zu wählen, ist mehr als der rechnerische Ressourcenbedarf zu berücksichtigen. Ein wichtiges Entwurfsziel ist Benutzbarkeit durch unerfahrene Benutzer. Um den Einstieg in den Umgang mit DynamicNodes zu erleichtern, ist das *data driven* Konzept vorzuziehen, da sich die Ausführung der Programme leichter nachvollziehen lässt.

Wenn mit DynamicNodes als visuellem Programmiersystem experimentiert wird, ist nicht immer eindeutig, ob Zwischenergebnisse, welche für ein evtl. gefordertes Endergebnis nicht nötig sind, nicht doch Informationen transportieren, welche für den Programmierer interessant sind. Dazu kommt, dass für den Entwurf von DynamicNodes weniger eine maximale Rechenleistung, als vielmehr Einfachheit und Benutzbarkeit im Vordergrund stehen. Diese Entwurfsziele sind mit dem *data driven* Konzept voraussichtlich leichter zu erreichen. Somit wird das *data driven* Konzept verwendet.

4.2.2 Vorbedingung

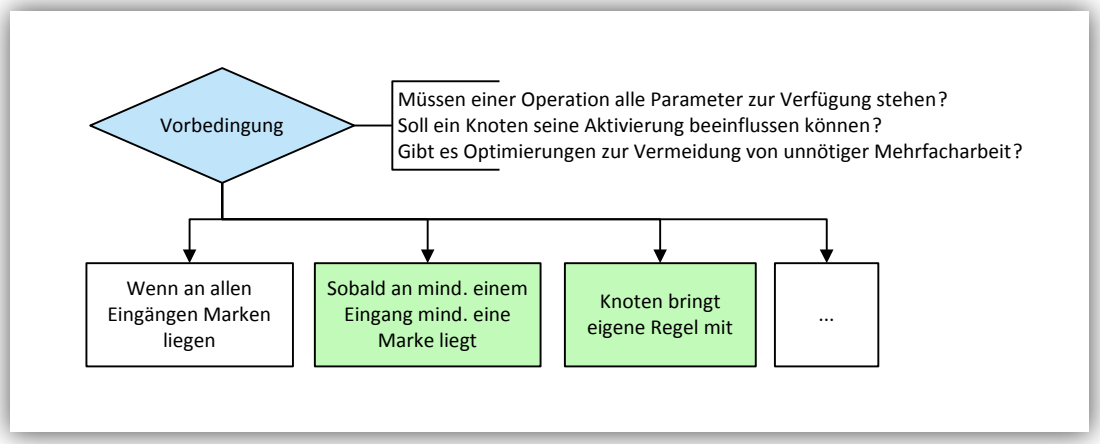


Abb. 22 Entwurfsentscheidungsmöglichkeiten für die Vorbedingung der Knotenaktivierung

Um die Aktivierung vollständig zu beschreiben, ist zu entscheiden wann eine Knotenoperation ausgeführt werden kann. Für eine mathematische Operation ist ebenso wie für die Operationsknoten in DynamicNodes-Programmen definiert, wie viele Operanden sie besitzt. Die Durchführung einer mathematischen Operation ist i.d.R. nur möglich, wenn alle Operanden verfügbar sind. Gibt es Operationen, die als DynamicNodes-Knoten implementiert werden können und welche nicht immer an allen Eingängen Marken benötigen? Diese Frage lässt sich nicht verneinen, da sich die Anwendungsgebiete von DynamicNodes schlecht abschätzen lassen. Um die Möglichkeiten bei der Entwicklung für Operationsknoten nicht einzuschränken, sollte angenommen werden, dass es solche Operationsknoten geben wird.

Also kann es für die Ausführung eines Knotens keine Bedingung sein, dass an allen Eingängen Marken anliegen. Folglich wird ein Knoten sofort aktiviert, sobald an einem seiner Eingänge eine neue Marke anliegt. Das ist zwar einfach, jedoch recht ineffizient, da ein Knoten mit mehreren Eingängen dann im Normalfall *mehrmals* aktiviert wird, um *ein* korrektes Ergebnis zu liefern. Eine zusätzliche Bedingung soll dieses Verhalten verbessern: *Ein Knoten wird nur aktiviert, wenn keiner der Knoten seines Vorbereichs mehr aktiv ist.*

Da diese Stelle der Flussmodellierung noch deutliches Optimierungspotenzial erkennen lässt, ist es sinnvoll den Knotenentwicklern eine Möglichkeit einzuräumen, die Aktivierungsbedingung mitzugestalten.

4.2.3 Nachbedingung

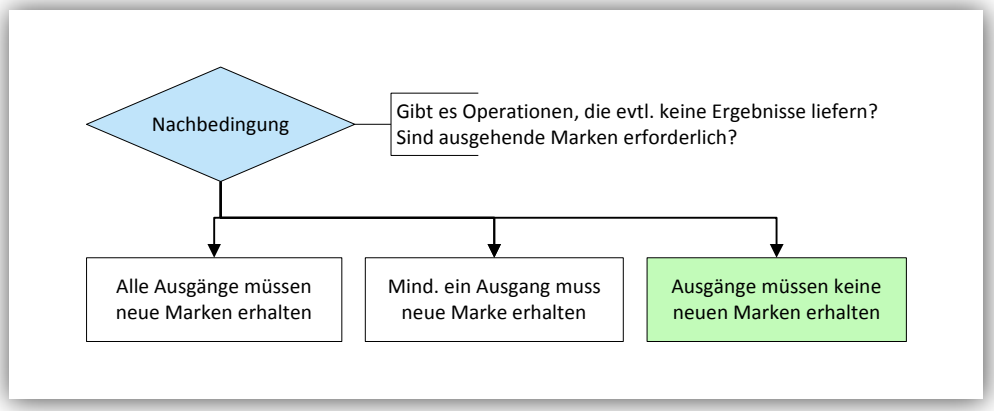


Abb. 23 Entwurfsentscheidungsmöglichkeiten für die Nachbedingung der Knotenaktivierung

Es gibt verschiedene Möglichkeiten für die Nachbedingung einer Knotenoperation. Da ein Knoten mehrere ausgehende Kanten haben kann, könnte eine Nachbedingung lauten: Alle ausgehenden Kanten müssen eine neue Marke erhalten haben. Oder: Mindestens eine ausgehende Kante muss eine neue Marke erhalten haben. Es muss also die Frage gestellt werden: Gibt es vielleicht Operationen, welche nicht immer an allen Ausgängen Ergebnisse liefern?

Es ist z.B. eine Verzweigungsoperation vorstellbar, die an einem Eingang A ein Datum und an einem zweiten Eingang B einen Wahrheitswert erwartet. Die Operation besitzt zwei Ausgänge C und D. Liegt

an Eingang B „wahr“ an, wird das Datum von Eingang A an Ausgang C weitergeleitet und D erhält nichts. Liegt an Eingang B „falsch“ an, wird das Datum von Eingang A an D weitergeleitet und C erhält nichts.

Die Frage über die Existenz von Operationsknoten, welche nicht an allen Ausgängen Ergebnisse liefern, muss also mit ja beantwortet werden. Nun stellt sich die Frage ob es Operationsknoten gibt, welche unter Umständen an keinem ihrer Ausgänge ein Ergebnis liefern.

Wird das Beispiel von oben vereinfacht und der Ausgang D entfernt, ergibt sich ein Schalter: Liegt am Eingang B „wahr“ an, wird das Datum von Eingang A an den Ausgang C weitergeleitet, sonst nicht.

Auch dies ein sinnvoller Operationsknoten und sollte so oder so ähnlich implementiert werden können. Also gibt es in diesem Sinne keine Nachbedingung für die Ausführung von Knotenoperationen. Es ist den Knotenentwicklern überlassen zu entscheiden, ob und wie viele Ausgänge eines Knoten während seiner Ausführung mit neuen Marken belegt werden.

4.2.4 Erzeugung von Knoten

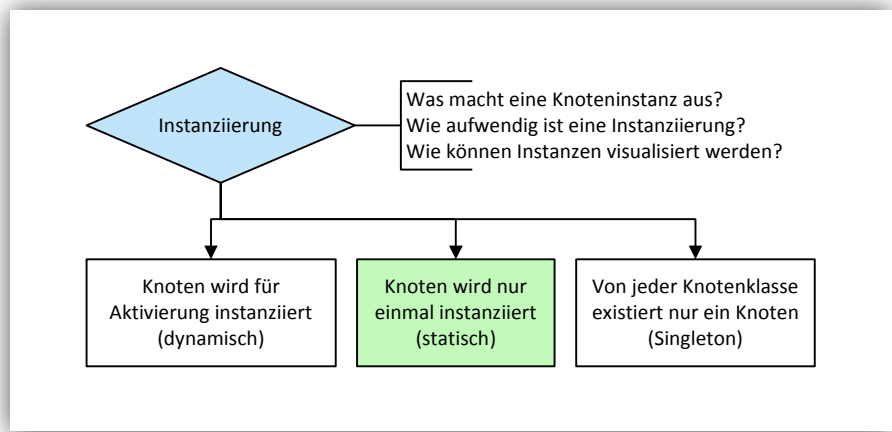


Abb. 24 Entwurfsentscheidungsmöglichkeiten für die Instanziierung von Operationsknoten

Ein DynamicNodes-Programm besteht i.d.R. aus unterschiedlichen Typen von Operationsknoten. Dabei können mehrere Knoten im Graph vom selben Typ sein.

Bei der Ausführung eines Graphen gibt es unterschiedliche Möglichkeiten den Programmcode für die Operation zur Verfügung zu stellen. Dabei ist zu berücksichtigen, dass DynamicNodes objektorientiert implementiert wird. Laut der Spezifikation soll die Operation eines Knotens durch eine Methode repräsentiert werden. Folglich ist der Programmcode eines Knotentyps als Klasse formuliert. Um Methoden einer Klasse aufrufen zu können, muss eine Instanz der Klasse (ein Objekt) erzeugt werden, auf welcher dann die Methode aufgerufen wird.

Nun könnte die Laufzeitumgebung von DynamicNodes, bei der Initialisierung für jeden Knotentyp, eine Klasse instanziieren (*singleton*), deren Methode immer dann aufgerufen würde, wenn ein Knoten dieses Typs im Graph aktiviert wird. Um mehrere Knoten des gleichen Typs parallel ausführen zu

können, dürfte diese Methode allerdings keine Zugriffe auf Instanz-Member durchführen, da sonst Konflikte mit anderen Ausführungspfaden entstehen würden. Neben dieser Einschränkung, die einer intuitiven Programmierung zuwider läuft, hebelt das Prinzip eines *singleton* die Vorteile objektorientierter Programmierung fast vollständig aus.

Das gegenteilige Extrem ist die Erzeugung einer Instanz des Knotentyps erst bei Aktivierung eines Knotens. Dieses Verfahren wird bei *dynamischen* Flussgraphen verwendet und ermöglicht die parallele Ausführung von Schleifen und Rekursionen (vergl. 2.1.2).

In 3.5.1 ist definiert, dass ein Knoten die Möglichkeit haben soll, seine grafische Ausgabe zu beeinflussen. Da DynamicNodes eine visuelle Entwicklungsumgebung sein soll, mit deren Hilfe der Benutzer während der Entwicklung direkt mit dem Graphen interagieren kann, müsste eine geeignete grafische Darstellung für einen mehrfach aktivierten Knoten entworfen werden. Allerdings ist es sehr schwierig eine beliebige Anzahl von Ausführungsinstanzen eines Knotens während seiner parallelen Ausführung zu visualisieren.

Ein weiterer Aspekt ist, dass Operationsknoten zwischen mehrmaliger Ausführung Informationen speichern können, welche die nachfolgende Ausführung beeinflusst. D.h. also dass Operationsknoten in DynamicNodes zustandsbehaftet sein können (vergl. 3.2). Ein solcher Zustand lässt sich schlecht verwalten, wenn die parallele Ausführung eines Knotens möglich ist.

Folglich muss ein Zwischenweg gewählt und jedem Knoten genau eine Instanz zugewiesen werden. Das dabei zur Anwendung kommende Konzept wird *statischer* Fluss genannt (vergl. 2.1.2). Dabei ist es nun ein Leichtes den evtl. vorhandenen Zustand einer Operation zu verwalten, da der Ausführungskontext einer Operation mit dem Kontext eines Knotens übereinstimmt. Nach dem objektorientierten Prinzip können nun Felder der Knoteninstanz für die Speicherung des Zustandes verwendet werden.

4.2.5 Organisation von Verbindungskapazität

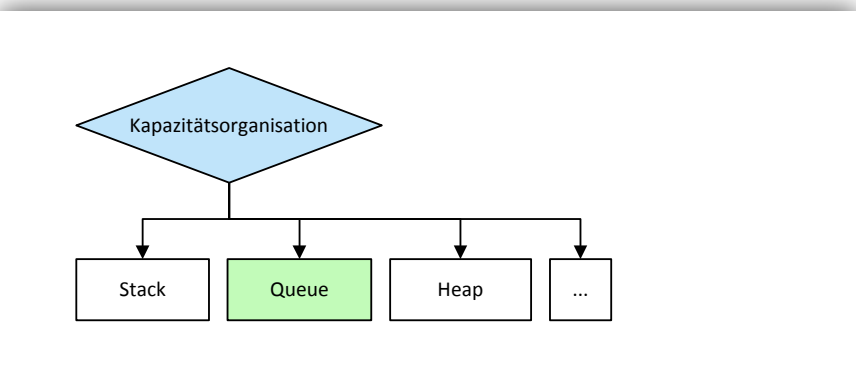


Abb. 25 Entwurfsentscheidungsmöglichkeiten für die Organsiation von Verbindungskapazität

Wenn ein Knoten schneller ausgehende Marken produziert als ein nachfolgender Knoten diese von seinem Eingang konsumiert, können evtl. mehrere Marken auf einer Verbindung liegen. Die mögliche

Menge an Marken auf einer Verbindung wird Verbindungskapazität genannt. Es gibt verschiedene Möglichkeiten diese Markenmenge zu organisieren.

Geht man davon aus, dass die Reihenfolge der Marken keine Rolle spielt, könnte man z.B. einen Heap verwenden und die Marken in ungeordneter Form darin ablegen und wieder entnehmen. Spielt die Reihenfolge jedoch eine Rolle, stellt sich die Frage welche Reihenfolge erforderlich ist, damit die Marken sinnvoll verarbeitet werden können. Für geordnete Speicher stehen z.B. ein Stack oder eine Queue zur Auswahl. Der Stack repräsentiert das Speicherkonzept LIFO[6] oder andersherum ausgedrückt FOLI[7]. Die Queue repräsentiert das Speicherkonzept FIFO[8] bzw. LILO[9].

Nimmt man an, dass eine Verbindung über längere Zeit mehrere Marken enthält, weil in etwa gleichschnell Marken am Ausgang des vorhergehenden Knotens erzeugt werden, wie sie am Eingang des nachfolgenden Knotens konsumiert werden, würde die erste Marke in einem Stack so lange auf ihre Verarbeitung warten, bis der Stack wieder vollständig abgearbeitet wird. Noch schlimmer: wenn ein Programm aus reinen Flussknoten besteht, könnte es vorkommen, dass das letzte Ergebnis aus den ersten Eingabedaten berechnet wird. Die Ergebnisse bei komplizierteren Programmen mit Schleifen oder Verzweigungen wären nur schlecht nachzuvollziehen.

Also, wird als Organisationform für die Verbindungskapazität die Queue gewählt. Sie sorgt dafür, dass ein leicht nachzuvollziehender Markenfluss entsteht.

4.2.6 Verbindungskapazität

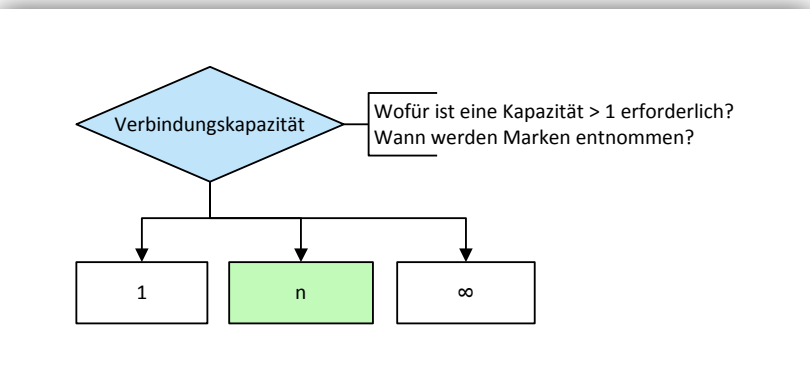


Abb. 26 Entwurfsentscheidungsmöglichkeiten für die Verbindungskapazität

Datenflusssysteme unterscheiden sich auch in der Kapazität ihrer Kanten. Es gibt Systeme, welche nur eine Marke auf jeder Kante erlauben. Andere Systeme erlauben eine Vielzahl von Marken auf jeder Kante.

[6] Last In First Out
[7] First In Last Out
[8] First In First Out
[9] Last In Last Out

Da DynamicNodes für die Entwicklung eines Programms so wenig Einschränkungen wie möglich machen soll, ist die optionale Verwendung einer Queue für eine Verbindung vorzusehen. Eine unendliche Größe für eine Queue ist nur theoretisch sinnvoll, kann jedoch nicht real implementiert werden. Demzufolge verwendet DynamicNodes Queues mit fester Länge, welche auch eine feste Kapazität von 1 möglich machen.

4.2.7 Unterscheidung von parallelen Marken

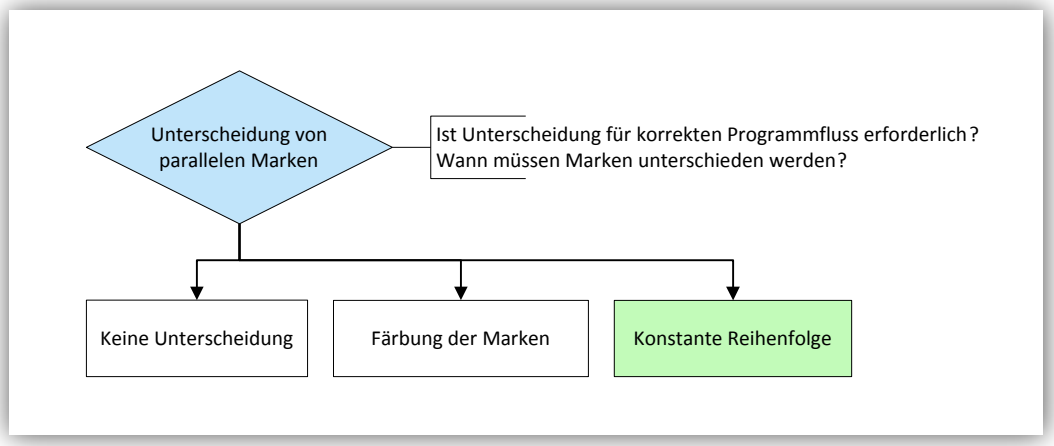


Abb. 27 Entwurfsentscheidungsmöglichkeiten für die Unterscheidung paralleler Marken

Eng im zusammen mit der Knotenerzeugung, hängt das Problem der Unterscheidung von parallelen Marken (vergl. 4.2.4). In einem dynamischen Fluss-System kann ein und derselbe Knoten zu einem Zeitpunkt mehrfach aktiviert sein. Das geschieht z.B. wenn ein Knoten die eingehenden Marken langsamer verarbeitet, als sie eintreffen. Das Ergebnis ist, dass auf den Ausgängen des Knotens Ergebnis-Marken in einer nicht definierten Reihenfolge abgelegt werden, welche von verschiedenen Ausführungsinstanzen desselben Knotens berechnet wurden.

Es gibt einen weiteren Fall in dem Marken aus unterschiedlichen Programmflüssen auf derselben Kante liegen können. Bei einer Schleife, wie in Abb. 28, können auf einer Kante viele Marken von unterschiedlichen Iterationen liegen.

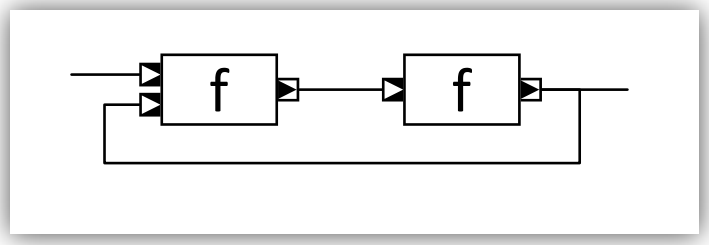


Abb. 28 Schleife

Das dabei entstehende Problem ist, dass jene Knoten, die Marken von mehreren Verbindungen konsumieren, welche wiederum mehrere Marken aus unterschiedlichen Programmflüssen tragen, entscheiden müssen, welche Marken als Eingabe sinnvoll zusammenpassen.

Diese Problematik tritt in dynamischen Flusssystemen auf. In 4.2.4 wurde festgelegt, dass DynamicNodes-Graphen statische Flussgraphen sind. Ein und derselbe Knoten wird also zu keiner Zeit mehrfach aktiv. Damit ist das Problem im Prinzip bereits gelöst, denn Marken können sich durch die in 4.2.5 festgelegten Queues nicht überholen. DynamicNodes-Knoten arbeiten in einem reinen Fließbandprinzip, wo alle eingehenden Marken in der Reihenfolge abgearbeitet werden, in der sie eintreffen. Eine explizite Unterscheidung von Marken ist durch die implizit vorhandene Reihenfolge nicht notwendig.

4.2.8 Lesekopie

Nachdem das DynamicNodes-Flusskonzept in den Abschnitten 4.2.1-4.2.6 nach den üblich Kriterien eines Flusssystems beschrieben wurde, soll noch eine Besonderheit beschrieben werden, welche bei den in Abschnitt 2.1.2 erwähnten Autoren nicht zu finden ist.

Um den Hintergrund zu klären, folgen einige Problemfälle, die bisher nicht gelöst sind:

- In 4.2.2 wurde festgelegt, dass ein Knoten auch ohne einen vollständigen Satz an neuen Eingabe-Marken aktiviert werden kann. Viele Operationen benötigen jedoch an allen Eingängen gültige Marken.
- Im experimentellen Umgang mit Operationsknoten ist es oft sinnvoll, lediglich die Marke *eines* Knoteneingangs zu aktualisieren, ohne *alle* Eingänge eines Knotens mit neuen Marken zu belegen. Dabei ist eine Aktivierung des Knotens mit den zuletzt verwendeten Eingangs-Marken auf allen übrigen Eingängen und der neuen Eingangs-Marke auf dem aktualisierten Eingang erwünscht.
- Wenn Knoten für eine höhere Abstraktionsebene entwickelt werden, z.B. ein Knoten, welcher eine Pixel-Matrix in eine Bild-Datei schreibt, ist es häufig sinnvoll, konstante Marken an Eingänge zu legen. Jener Knoten könnte z.B. neben dem Eingang, welcher die Pixel-Matrix entgegen nimmt, einen Eingang besitzen, welcher den Dateityp entgegennimmt. Nun ist es selten der Fall, dass der Dateityp von anderen Knoten berechnet werden muss. Ein Benutzer möchte vielmehr zur Entwicklungszeit eines Programms mit diesem Knoten festlegen, welcher Dateityp verwendet werden soll. Dazu muss er die Möglichkeit haben, eine Marke an einen offenen Eingang zu platzieren, welche bei der Ausführung des Knoten nicht verschwindet.
- In dem visuellen Editor wäre es hilfreich wenn man sowohl wenn der Graph arbeitet als auch dann wenn keiner der Knoten mehr aktiv ist (der Graph tot ist) eine Möglichkeit hätte, die zuletzt an den Eingängen anliegenden Marken einzusehen.

Die Lösung für die oben genannten Probleme ist eine sog. „Lesekopie“. Sowohl jeder Eingang als auch jeder Ausgang erhält den Speicherplatz für eine einzelne Marke. Im Eingang wird dieser Speicherplatz zusätzlich zu der evtl. vorhandenen Queue eingerichtet. Der Ausgang besitzt nur diesen Speicherplatz.

4.2.9 Marken

Die Marken eines Flusssystems sind Transportcontainer für die Informationen, welche durch das System verarbeitet werden sollen. Ein DynamicNodes-Graph soll Konstanten, variable Werte und Referenzen in Marken transportieren können.

Steuermarken

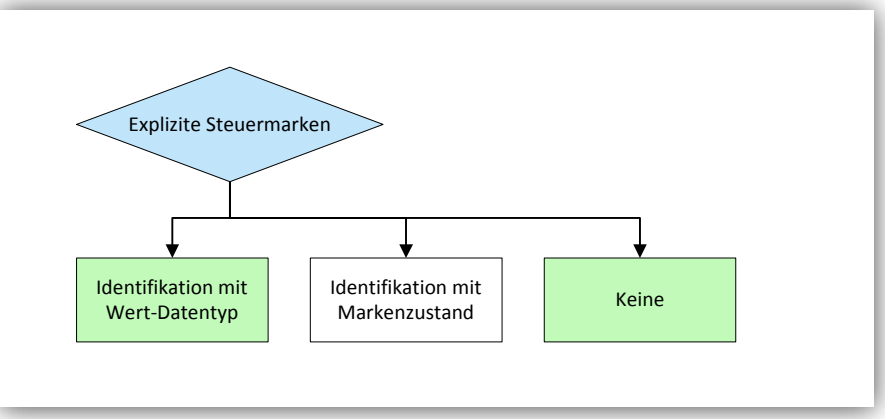


Abb. 29 Entwurfsentscheidungsmöglichkeiten für die Verwendung von Steuermarken

In einigen Flusssystemen wird zwischen Daten- und Steuermarken unterschieden. Dabei werden Steuermarken dazu eingesetzt Schalter, Verzweigungen oder Auswahlen zu steuern.

Es gibt zwei Möglichkeiten die Datenstruktur einer Marke so anzulegen, dass Steuermarken und Datenmarken die gleiche Datenstruktur verwenden, jedoch leicht zu unterscheiden sind. Die einfachste Möglichkeit ist es, eine Gruppe von Typen als Steuertypen zu deklarieren. Jede Marke, welche einen Wert mit einem Datentyp aus der oben genannten Gruppe besitzt, ist eine Steuermarke. Die zweite Möglichkeit ist, die möglichen Zustände um einen Zustand „Steuermarke“ zu ergänzen (vergl. S. 61). Jede Marke mit dem Zustand „Steuermarke“ ist demzufolge eine Steuermarke.

Für DynamicNodes stellt sich die Frage, ob Steuermarken explizit vorgegeben werden sollten oder nicht. DynamicNodes soll dem Knotenentwickler bei der Gestaltung des Knotenverhaltens so viel Freiraum wie möglich lassen. Da Marken in DynamicNodes eine Vielzahl von verschiedenen Datentypen transportieren können, u.a. auch Konstanten, kann jeder Knotenentwickler selbst eine Menge von Steuerkonstanten als Datentyp definieren (erste Möglichkeit) und Eingänge so erzeugen, dass sie ausschließlich Marken mit diesem Datentyp entgegennehmen.

Die Datenstruktur der Marken selbst muss also keine explizite Unterstützung für Steuermarken enthalten.

Markenzustände

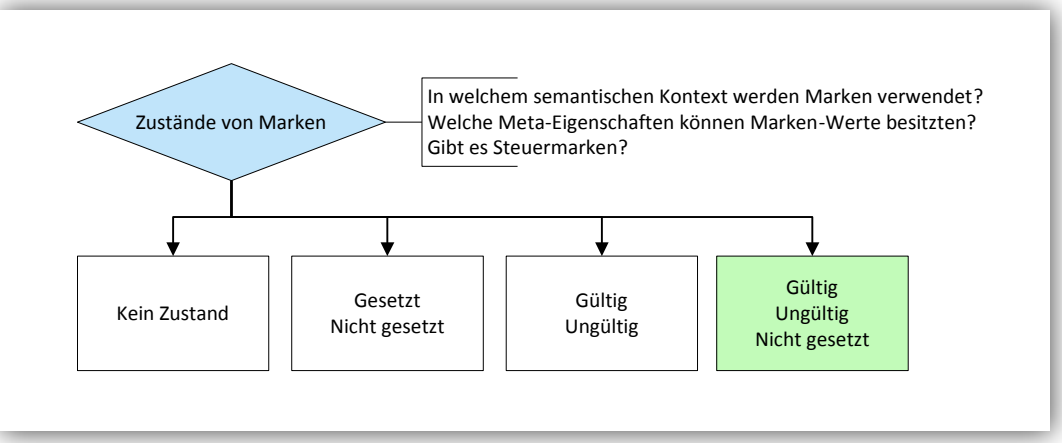


Abb. 30 Entwurfsentscheidungsmöglichkeiten für Markenzustände

Jeder Anschluss besitzt eine Lesekopie und damit zu jeder Zeit eine Marke, welche gelesen werden kann (vergl. 4.2.8). Deshalb müssen Marken die Fähigkeit besitzen, anzuzeigen wenn keine Daten vorhanden sind.

Der Markenwert unterstützt mit den Datentypen Klassen „by literal“, „by value“ und „by reference“ eine bestimmte Wertedomäne. Darin enthalten ist auch der leere Verweis (in C# `null`). Um den Zustand eines Markenwertes angeben zu können, also mit einer Art Metadatum zu versehen, wird Speicherplatz außerhalb der im Markenwert implizierten Domäne benötigt. Die Marke muss also ein zusätzliches Feld für die Aufnahme eines Markenzustandes erhalten.

Nun stellt sich die Frage, ob „Daten vorhanden“ und „Daten nicht vorhanden“ die einzigen Zustände sind, welche eine Marke kennzeichnen können.

Um der Entwicklung von Knotenoperationen mehr Spielraum einzuräumen, soll der Aspekt der Gültigkeit von Werten hinzugefügt werden. Beispielsweise könnte eine Knotenoperation, welche eine Addition ausführt, bei Überlauf eine Marke ohne Daten weitergeben, was aber zwangsläufig zum Abbruch der nachfolgenden Verarbeitung führen würde. Dieser Effekt ist nicht immer erwünscht. Besser ist es dem Knotenentwickler eine Möglichkeit zu geben, Ergebnisse welche noch einigermaßen Sinn ergeben, jedoch nicht korrekt sind, als solche zu markieren.

Eine Marke besitzt also ein Zustandsfeld, welches die drei Zustände „Daten gültig“, „Daten ungültig“ und „Daten nicht vorhanden“ annehmen kann. Zur einfachen Handhabung werden diese Zustände im Folgenden „Gültig“ (`Valid`), „Ungültig“ (`Invalid`) und „Nicht gesetzt“ (`NotSet`) genannt.

4.2.10 Datenfluss

Als Abschluss dieses Kapitels, soll noch einmal zusammenfassend veranschaulicht werden nach welchem Algorithmus der Datenfluss arbeitet.

Die Beschreibung wird dabei zunächst aus der Perspektive eines Ausgangs und anschließend aus der Perspektive eines Eingangs beschrieben.

Ausgang

Bei Initialisierung des Knotens wird in der Lesekopie jedes Ausgangs eine Marke ohne Wert abgelegt.

Wenn ein Knoten während seiner Arbeit eine Marke auf einen Ausgang legt, wird diese als Lesekopie im Ausgang gespeichert und als neue Marke gekennzeichnet. Aktualisiert der Knoten während seiner Arbeit einen Ausgang mehrmals, wird die Lesekopie überschrieben und die zuvor abgelegte Marke verfällt.

Wenn der Knoten seine Arbeit beendet hat, werden alle Ausgänge, welche neue Marken besitzen, aufgefordert, die als Lesekopie gespeicherte Marke weiterzugeben. Dies ist nur den Ausgängen möglich, welche mit mind. einem Eingang verbunden sind. Alle verbundenen Eingänge eines Ausgangs erhalten eine Kopie der Marke auf dem Ausgang. Dabei bleibt sowohl bei den verbundenen als auch bei den nicht verbundenen Ausgängen die Marke als Lesekopie erhalten. Anschließend werden alle Knoten, welche durch einen Eingang mit den Ausgängen des Knotens verbunden sind, aktiviert und die als neu gekennzeichneten Lesekopien auf den Ausgängen als nicht neu gekennzeichnet.

Werden eine oder alle Verbindungen eines Ausgangs getrennt, bleibt die Lesekopie erhalten.

Wird ein Ausgang mit einem Eingang verbunden, wird die Lesekopie des Ausgangs unverzüglich an den Eingang weitergegeben und der Knoten des Eingangs aktiviert. Auch dabei bleibt die Lesekopie des Ausgangs erhalten.

Die Lesekopie kann jederzeit vom Editor oder anderen Komponenten gelesen werden.

Eingang

Bei Initialisierung eines Knotens wird in der Lesekopie eines jeden Eingangs eine Marke ohne Wert abgelegt. Oder der Knoten selbst belegt während der Initialisierung einen Eingang mit einer Standard-Marke.

Wenn ein Eingang von einem verbundenen Ausgang eine Marke erhält, wird diese zu allererst als Lesekopie des Eingangs gespeichert. Als nächstes wird eine Kopie der Marke in die Queue des Eingangs gelegt. Soll der Eingang keine Queue besitzen, kann diese auf die Länge 1 gesetzt werden. Ist die Queue voll, wird die älteste Marke am Ende der Queue entfernt, bevor die neue Marke in die Queue gelegt wird. Die entfernte Marke verfällt. Die Lesekopie eines Eingangs enthält somit immer die zuletzt auf den Eingang gelegte Marke.

Sobald ein Eingang eine neue Marke erhält, wird er mit einem Flag gekennzeichnet. Anhand dieser Kennzeichnung kann ein Knoten seine Eingänge auf neue Marken überprüfen.

Bevor die Arbeitsmethode eines Knotens aufgerufen wird, werden von allen Eingängen die aktuellen Eingangs-Marken gesammelt und der Arbeitsmethode zugänglich gemacht. Von Eingängen, deren Queue leer ist, wird die Lesekopie eingesammelt. Von Eingängen, deren Queue nicht leer ist, wird die älteste Marke am Ende der Queue eingesammelt und dabei aus der Queue entfernt. Wird die letzte Marke aus einer Queue entfernt, wird die Kennzeichnung des Eingangs, mit deren Hilfe der Knoten erkennen kann, dass eine neue Marke verfügbar ist, entfernt.

Wird ein Eingang von einem Ausgang getrennt, wird eine Marke ohne Wert auf den Eingang gelegt und der Knoten des Eingangs wird aktiviert. Ist ein Eingang nicht mit einem Ausgang verbunden, kann der Editor oder ein andere Komponente eine neue Marke auf den Eingang legen und evtl. den Knoten aktivieren.

Die Lesekopie des Eingangs kann jederzeit vom Editor oder einer anderen Komponente gelesen werden.

4.3 Formale Definition

An dieser Stelle soll das Ergebnis des Entwurfs, für das Datenflusskonzept von DynamicNodes-Programmen, formal zusammengefasst werden.

4.3.1 Programmstruktur

Zunächst wird die Struktur des Graphen definiert, ohne die Flusseigenschaften zu berücksichtigen.

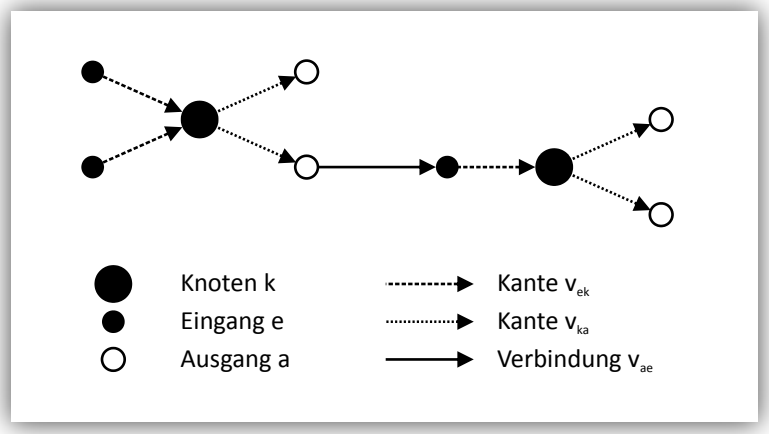


Abb. 31 Formaler DynamicNodes-Graph

Der Graph G

Ein Graph in DynamicNodes ist definiert als ein Tupel $G = (K, E, A, V, q, m, w, p, t)$, wobei K die Menge aller Knoten im Graphen, E die Menge aller Eingänge, A die Menge aller Ausgänge und V die Menge aller Kanten im Graph ist. Es gilt $(K \cap E = \emptyset) \wedge (K \cap A = \emptyset) \wedge (E \cap A = \emptyset)$.

Im Folgenden ist, wenn nicht anders angegeben, anzunehmen, dass $e \in E$, $a \in A$ und $k \in K$.

Die übrigen Elemente von G werden im Abschnitt 0 erläutert.

Die Kanten V

Die Menge der Kanten V lässt sich in die Partition $\mathcal{V} = \{V_{ek}, V_{ka}, V_{ae}\}$ aufteilen. Dabei ist V_{ek} die Menge aller Kanten zwischen Eingängen und Knoten, V_{ka} ist die Menge aller Kanten zwischen Knoten und Ausgängen und V_{ae} ist die Menge aller Kanten zwischen Ein- und Ausgängen, welche auch Verbindungen genannt werden.

Jeder Graph besitzt eine auf der Kantenmenge V definierte Abbildung B, welche jeder Kante ein Paar $(x, y) \in (E \times K) \cup (K \times A) \cup (A \times E)$ zuordnet. Diese Abbildung kann auch als Zusammenfassung der folgenden drei Abbildungen aufgefasst werden. $k \in K$

$$\mathrm{B}_{ek}: V_{ek} \mapsto (e, k)$$
$$\mathrm{B}_{ka}: V_{ka} \mapsto (k, a)$$
$$\mathrm{B}_{ae}: V_{ae} \mapsto (a, e)$$

Anschaulich heißt dies, dass B jeder Kante entweder ein Paar aus Eingang und Knoten, ein Paar aus Knoten und Eingang oder ein Paar aus Ausgang und Eingang zuordnet.

Für den Ausdruck $\exists\ v \in V \mid \mathrm{B}(v) = (x, y)$ wird im Folgenden auch die vereinfachte Schreibweise xVy verwendet.

Für jeden Eingang gibt es nur genau eine Kante, welche in einem Knoten endet.

$$\forall\ e : |\{\ v_{ek} \in V_{ek} \mid \mathrm{B}_{ek}(v_{ek}) = (e, k)\ \}| = 1$$

In jedem Ausgang endet genau eine Kante, die von einem Knoten ausgeht.

$$\forall\ a : |\{\ v_{ka} \in V_{ka} \mid \mathrm{B}_{ka}(v_{ka}) = (k, a)\ \}| = 1$$

Eine Kante $v_{ae} \in V_{ae}$ zwischen einem Ausgang und einem Eingang wird Verbindung genannt.

Von jedem Ausgang können mehrere Verbindungen ausgehen.

$$\forall\ a : |\{\ v_{ae} \in V_{ae} \mid \mathrm{B}_{ae}(v_{ae}) = (a, e)\ \}| \geq 0$$

In einem Eingang kann höchstens eine Verbindung enden.

$$\forall\ e : |\{\ v_{ae} \in V_{ae} \mid \mathrm{B}_{ae}(v_{ae}) = (a, e)\ \}| = \{0,1\}$$

Durch das Tupel (K, E, A, V) ist ein 3-partiter gerichteter Graph definiert.

Eigenschaften von Knoten und Kanten

Die Kapazitätsfunktion q bildet jeden Eingang auf eine Warteschlangengröße ab.

$$q : E \mapsto \mathbb{N} \cup \{\infty\}$$

Die Funktion m gibt für jeden Ein- und Ausgang die Anzahl der Marken an.

$$m : E \cup A \mapsto \mathbb{N}$$

Die Funktion w gibt für jeden Knoten an, ob er arbeitet oder nicht.

$$w : K \mapsto \{\text{"arbeitend"}, \text{"wartend"}\}$$

Eine Marke ist ein Paar $M = (s, d)$ mit dem Markenstatus $s \in \{\text{"nicht gesetzt"}, \text{"ungültig"}, \text{"gültig"}\}$ und dem Markenwert $d \in D$. Die Menge der möglichen Werte ist $D = D_r \cup D_v \cup D_c$, wobei D_r die Klasse der Verweisdatentypen, D_v die Klasse der Wertdatentypen und D_c die Klasse der Konstanten-Datentypen ist.

Jeder Ein- und Ausgang hat unabhängig von seiner Kapazität immer eine sog. „Lesekopie“ der letzten Marke, welche auf den Anschluss gelegt wurde. Auch wenn $m(e) = 0; e \in E$ kann die Lesekopie eines Eingangs als Eingabe für die Ausführung eines Knotens verwendet werden.

Ob ein Knoten parallel oder seriell arbeitet ist durch die Funktion p definiert.

$$p : K \mapsto \{\text{"parallel"}, \text{"seriell"}\}$$

In welchem Ausführungspfad ein Knoten arbeitet, ist durch die Funktion t definiert, welche jeden Knoten auf eine natürliche Zahl abbildet.

$$t : K \mapsto \mathbb{N}$$

Das Ergebnis von $t(k)$ ist dann eine gültige Identifikation für einen Ausführungspfad, wenn gilt

$$w(k) = \text{"arbeitend"}$$

Hilfsfunktionen

Einige Funktionen helfen weitere Definitionen und Regeln zu vereinfachen:

Knoten eines Eingangs:	$K(e) = k : eVk$
Knoten eines Ausgangs:	$K(a) = k : kVa$
Eingänge eines Knotens:	$E(k) = \{e \mid eVk\}$
Ausgänge eines Knotens:	$A(k) = \{a \mid kVa\}$
Eingänge eines Knotens mit Marken:	$E^+(k) = \{e \in E_k(k) \mid m(e) > 0\}$
Eingänge eines Knotens ohne Marken:	$E^0(k) = \{e \in E_k(k) \mid m(e) = 0\}$
Alle mit einem Ausgang verbundenen Eingänge:	$E(a) = \{e \in E \mid eVa\}$
Der mit einem Eingang verbundene Ausgang:	$A(e) = a \in A : (eVa)$

Wie leicht ersichtlich ist, sind $E^+(k)$ und $E^0(k)$ disjunkt.

$$E^+(k) \cap E^0(k) = \emptyset$$

Vor- und Nachbereich eines Knotens

Der Vorbereich eines Knotens ${}^\star k$ ist die Menge aller Knoten, von denen mindestens ein Ausgang mit mindestens einem Eingang von k verbunden ist.

$${}^\star k = \{k' \in K \mid a'Ve'\}; \left(a' \in A_k(k'), e' \in E_k(k)\right)$$

Der Nachbereich eines Knotens $k^\star$ ist die Menge aller Knoten, von denen mindestens ein Eingang mit einem der Ausgänge von k verbunden ist.

$$k^\star = \{k' \in K \mid a'Ve'\}; \left(a' \in A_k(k), e' \in E_k(k')\right)$$

4.3.2 Laufzeitverhalten

Aktivierung eines Knotens

Ein Knoten kann zu jedem Zeitpunkt aktiviert werden.

Ein Knoten wird durch die Laufzeitumgebung oder einen anderen Knoten aktiviert. Wenn ein Knoten k aktiviert wird, müssen die folgenden Vorbedingungen erfüllt sein, damit der Knoten arbeiten darf.

Ein Knoten k darf arbeiten wenn gilt:

Der Knoten darf nicht aktiv sein.

$$w(k) = \text{"wartend"}$$

Kein Knoten k' aus dem Vorbereich von k darf in einem fremden Ausführungspfad aktiv sein.

$$\forall\, k' \in {}^{\star}k : w(k') = \text{"wartend"} \vee t(k) = t(k')$$

Wenn der Knoten aktiviert wurde, jedoch nicht arbeiten darf, wird er deaktiviert.

Arbeitsphasen

Die Arbeit von Knoten mit $p(k) = \text{"parallel"}$ beginnt in der gleichen Reihenfolge, wie deren Aktivierung. Oder anders ausgedrückt, es gibt keinen Zeitpunkt t, zu dem ein Knoten k_1 welcher zum Zeitpunkt t_1 aktiviert wurde, seine Arbeit begonnen hat und zu dem ein Knoten k_2 nach seiner Aktivierung zum Zeitpunkt t_2 seine Arbeit noch nicht begonnen hat, wenn $t_2 < t_1 < t$.

Ein Knoten k arbeitet in drei Phasen. Der Ladephase, der Ausführungsphase und der Weitergabephase.

Ladephase

In der Ladephase werden die Marken $M_e(e); \big(e \in Eingaenge(k)\big)$ für die Ausführungsphase von den Eingängen geholt. Für $M_e(e)$ kann auch M_e geschrieben werden.

$$M_e = \begin{cases} e \in E^0(k) : \text{ Lesekopie von } e \text{ wird kopiert} \\ e \in E^+(k) : \text{älteste Marke auf } e \text{ wird entfernt} \end{cases}$$

Dabei ändert sich die Anzahl der Marken auf jedem Eingang e wie folgt.

$$m'(e) = \begin{cases} m(e) = 0 : 0 \\ m(e) > 0 : m(e) - 1 \end{cases}$$

Ausführungsphase

In der Ausführungsphase wird der Programmcode des Knotens ausgeführt, welcher die Eingabemarken M_e verarbeitet. Dabei hat der Programmcode die Möglichkeit auf einer Untermenge $A' \subseteq A(k)$ der Ausgänge neue Marken $M_a(a); (a \in A')$ zu platzieren. Für $M_a(a)$ kann auch M_a geschrieben werden. Wenn der Programmcode auf einem der Ausgänge während einer Ausführungsphase mehrere Marken platziert, bleibt nur die jüngste Marke erhalten.

Die Anzahl der Marken auf den Ausgängen ändert sich dabei wie folgt.

$$m'(a) = \begin{cases} a \in A' : 1 \\ a \notin A' : m(a) \end{cases}$$

Weitergabephase

In der Weitergabephase, werden alle Marken auf den Ausgängen an die verbundenen Eingänge weitergegeben. Und alle Knoten, deren Eingänge bei diesem Prozess neue Marken erhalten haben werden aktiviert.

$$A' = \{a \in A(k) \mid m(a) > 0\}$$
$$E' = \{e \in E \mid \exists\, a \in A' : A(e) = a\}$$
$$K' = \{k \in K \mid \{E(k) \cap E'\} \neq \emptyset\}$$

Die Anzahl der Marken auf den Ein- und Ausgängen ändert sich dabei nach dem Prinzip

$$m'(a) = 0; (a \in A')$$
$$m'(e) = \begin{cases} m(e) < q(e) : m(e) + 1 \\ m(e) = q(e) : m(e) \end{cases}; (e \in E')$$

Wenn die Kapazität des Eingangs $q(e)$ erschöpft ist, wird die älteste Marke in seiner Warteschlange entfernt.

Anschließend werden alle Knoten der Menge K' aktiviert.

4.4 Datenmodell

Auf der Basis des theoretischen Entwurfs für das Flusskonzept in Abschnitt 4.2 und 4.3 kann das Datenmodell entwickelt werden. Dabei soll besonderes Augenmerk auf die Spezifika der Laufzeitbibliothek .NET 2.0 gelegt werden. Das Datenmodell wird in der Implementierung mit C# umgesetzt und muss dazu auf das objektorientierte Programmierparadigma abgebildet werden.

4.4.1 Marken

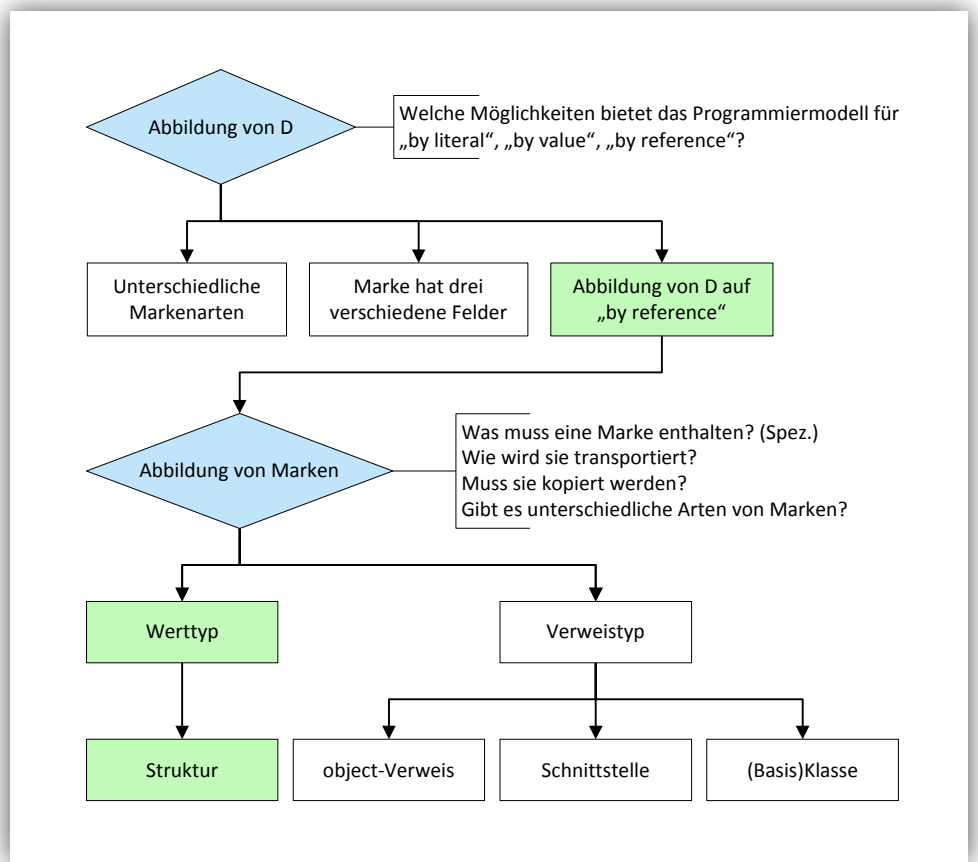


Abb. 32 Entwurfsentscheidungsmöglichkeiten für die Abbildung von Marken

Es soll einmal mit den Marken begonnen werden. Eine Marke besteht aus zwei Bestandteilen: dem Markenwert und dem Markenzustand (vergl. 4.2.9). Der Markenwert muss Daten aus allen Datentypenklassen enthalten können, die in der in 4.3.1 definierten Menge D enthaltenen sind. Der Zustand der Marke s ist ein Wert aus einer kleinen Menge Konstanten, welche ebenfalls in 4.3.1 definiert ist.

Abbildung von Konstanten, Werten und Verweisen in .NET

Eine wichtige Frage ist, wie die Werte der drei Typen-Klassen aus D in einer .NET-kompatiblen Datenstruktur abgelegt werden können.

Was sich zunächst feststellen lässt, ist dass sich die Klasse der konstanten Datentypen D_c leicht auf die Klasse der Wert-Datentypen D_v abbilden lässt, indem innerhalb jeder konstanten Typenklasse jeder Konstante eine eindeutige Ganzzahl zugeordnet wird. Diese Ganzzahlen lassen sich technisch z.B. als Datentyp `System.Int32` darstellen und würden damit zur Wert-Datentypen-Klasse gehören. Eine künstliche Abbildung ist jedoch in C# nicht notwendig, da dort bereits ein Sprachmittel für die Definition von Konstanten-Mengen existiert. Das Sprachmittel heißt Aufzählung oder engl. Enumeration (`enum`) und erlaubt die Definition einer Menge benannter Konstanten, welche als Datentyp zusammengefasst sind und intern auf `System.Int32` abgebildet werden.

Sobald also eine Konstantenmenge (ein Element der Menge D_c) als Datentyp benötigt wird, kann diese Menge als Aufzählung definiert werden.

Nun stellt sich die Frage, wie in einer einzigen Datenstruktur unterschiedliche Arten von Wert-Datentypen gespeichert werden können. Da für Wert-Datentypen Speicherplatz auf dem Stack reserviert wird, muss bei Deklaration eines Feldes mit Wert-Datentyp die Größe des benötigten Speicherplatzes bekannt sein. Es wird jedoch eine Möglichkeit gesucht eine beliebige Anzahl unterschiedlicher Wert-Datentypen mit unbekanntem Speicherplatzbedarf in einem Feld zu speichern. Das ist offensichtlich mit der Deklaration eines Wert-Datentyp-Feldes nicht möglich.

Aber es gibt dennoch eine einfache Lösung. Da in .NET alle Wert-Datentypen (mit dem Schlüsselwort `struct` deklarierte Datenstrukturen) kompatibel mit dem Verweis-Datentyp `System.Object` sind und sogar Schnittstellen implementieren können, ist es mit C# kein Problem einen Wert-Datentyp einem Feld zuzuweisen, welches für einen kompatiblen Verweis-Datentyp deklariert wurde. Das funktioniert immer mit Feldern vom Typ `System.Object`. Der Wert-Datentyp wird dafür automatisch in einen Verweis konvertiert, wobei die Daten vom Stack in Heap verschoben werden. Dieser Mechanismus nennt sich In-Boxing. Wird nun der Verweis auf den Wert-Datentyp wieder einem Feld zugewiesen, welches für diesen Wert-Datentyp deklariert wurde, werden die Daten vom Heap zurück auf den Stack kopiert. Dieser Mechanismus nennt sich Out-Boxing.

Dieses Vorgehen soll noch einmal kurz an einem Beispiel verdeutlicht werden. Zunächst wird eine Konstantenmenge definiert und ein Feld deklariert, welches eine Konstante aufnimmt:

```
enum Wahrheitswerte { Wahr, Falsch }
Wahrheitswerte b = Wahrheitswerte.False;
```

Als Wert-Datentyp wird eine Fließkommazahl verwendet:

```
double pi = 3.14159265358979323;
```

Und nun werden noch ein Verweis-Datentyp und ein entsprechendes Feld deklariert:

```
class Rechner
{
    public int wert;
    public int Addiere(int summand)
    {
        wert += summand;
        return wert;
    }
}
Rechner r = new Rechner();
```

Nun enthält die Datenstruktur der Marke ein Feld von Datentyp `System.Object`:

```
object wert;
```

Durch die oben beschriebenen Automatismen sind folgende Zuweisungen absolut zulässig:

```
wert = b;                    // Auto-In-Boxing
b = (Wahrheitswerte)wert;    // Auto-Out-Boxing und explizites casten
wert = pi;                   // Auto-In-Boxing
pi = (double)wert;           // Auto-Out-Boxing und explizites casten
wert = r;                    // Normale Zuweisung eines Verweises
r = (Rechner)wert;           // explizites casten
```

Es ist also durch den Mechanismus des automatischen In- und Out-Boxing möglich, ein Feld vom Typ `System.Object` anzulegen und diesem beliebige Verweise, beliebige Werte und beliebige Konstanten zuzuweisen. Bei diesem Vorgehen werden Konstanten automatisch auf Werte und Werte automatisch auf Verweise abgebildet.

Datenstruktur einer Marke

Für die Datenstruktur der Marke kommen drei technische Abbildungen in Frage. Ein einfaches Verweisfeld vom Type `System.Object` kommt nicht in Frage, da eine Marke dann keinen Zustand besitzen könnte.

Die erste Möglichkeit ist eine Schnittstelle, welche den Zugriff auf den Wert der Marke und den Zustand der Marke beschreibt. Zusätzlich müsste die Schnittstelle eine Methode zum Kopieren der Marke definieren, da eine Marke bei der Übergabe von Ausgang zu Eingang und beim Lesen der Lesekopie kopiert werden soll. Eine solche Schnittstelle ist sinnvoll, wenn zu erwarten ist, dass es mehrere unterschiedliche Implementierungen von Marken geben wird, welche von keiner gemeinsamen Basisklasse abstammen. Der Aufwand für die Implementierung von Schnittstelle, konkreter Klasse und Kopiermethode scheint für eine Marke unnötig hoch.

Eine konkrete Klasse, die evtl. auch als Basisklasse für spezialisierte Marken verwendet werden könnte, ist eine zweite Möglichkeit. Diese Klasse könnte Felder für Markenwert und Zustand deklarieren und den kontrollierten Zugriff darauf erlauben. Eine Kopiermethode könnte virtuell deklariert werden, so dass eine ableitende Klasse den Kopiermechanismus erweitern könnte. Jedoch ist immer noch unklar, ob ein Verweis-Datentyp für eine Marke angebracht ist.

Folgende Kriterien sind bei der Entscheidung zwischen Verweis-Datentyp und Wert-Datentyp ausschlaggebend:

- Verweis-Datentypen werden im Heap verwaltet
- Die Übergabe eines Verweis-Datentyps kopiert lediglich den Verweis, nicht die Datenstruktur
- Wert-Datentypen werden auf dem Stack verwaltet
- Die Übergabe eines Wert-Datentyps kopiert automatisch die Datenstruktur
- Wert-Datentypen können durch automatisches In- und Out-Boxing, Feldern mit einem kompatiblen Verweis-Datentyp zugewiesen werden (In- und Out-Boxing ist recht langsam)
- Marken werden häufig erzeugt, übergeben, zerstört und von verschiedenen Knoten bearbeitet

Der Vorteil von Verweis-Datentypen ist ein geringerer Speicherplatzverbrauch. Denn für jede Marke, welche von Knoten zum Ausgang, anschließend zu evtl. mehreren Eingängen und dann zu mehreren Knoten transportiert würde, wäre nur einmal Speicherplatz nötig. Die Datenstruktur der Marke besteht aus einem Feld für den Zustand, welcher intern auf `System.Int32` abgebildet wird, und einem Verweisfeld auf den Markenwert, welcher ebenfalls 4 Byte einnimmt. Der Verweis auf eine Marke würde auch 4 Byte kosten. Somit ist die Markenstruktur gerade einmal 8 Byte groß.

Ein Wert-Datentyp wird automatisch kopiert, kann schnell auf dem Stack erzeugt und als Parameter „by value“ übergeben werden. Die Zerstörung erfordert kein Eingreifen des GC. Der Speicherplatzverbrauch ist nur geringfügig größer. Diese Vorteile sprechen deutlich für die Deklaration der Marke als Wert-Datentyp.

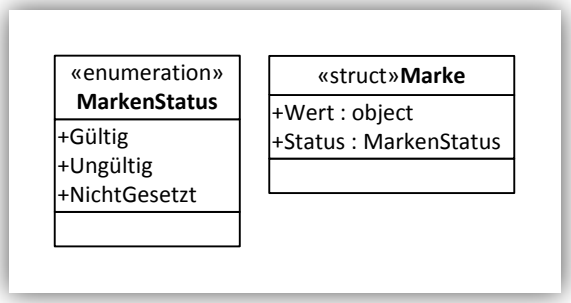


Abb. 33 UML-Diagramm für Marke und Markenstatus

Die Implementierung von in C# könnte demnach in etwa wie folgt aussehen:

```
public enum MarkenStatus
{
    Gültig, Ungültig, NichtGesetzt
}

public struct Marke
{
    public object wert;
    public MarkenStatus status;
}
```

4.4.2 Graph und Knoten

Für die Ausführung des Graphen muss seine Struktur in einer geeigneten Weise im Arbeitsspeicher vorgehalten werden. Dafür gibt es zwei unterschiedliche Ansätze(Creutzburg, 2003). Der eine ist die Adjazenzmatrix und der andere ist eine verkettete Speicherung.

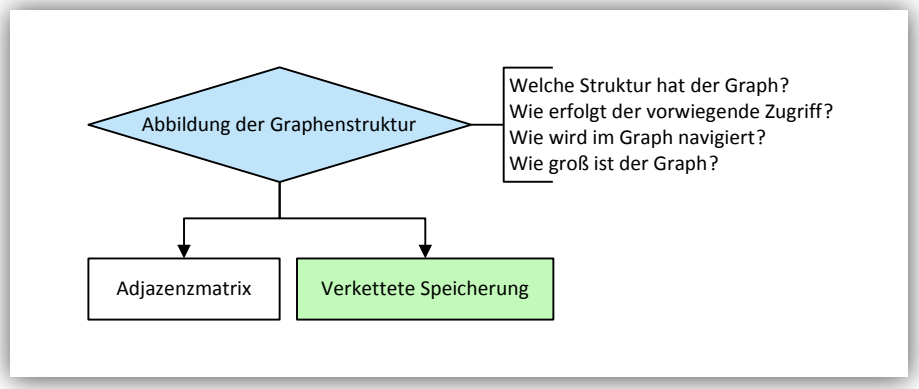


Abb. 34 Entwurfsentscheidungsmöglichkeiten für die Speicherung der Graphenstruktur

Die Adjazenzmatrix ist eine Tabelle mit soviel Spalten und Zeilen, wie der zu speichernde Graph Knoten enthält. Jedem Knoten wird sowohl genau eine Zeile als auch genau eine Spalte zugeordnet. In den Feldern der Tabelle wird nun die Anzahl der Verbindungen eingetragen, welche von dem Knoten der Zeile zu dem Knoten der Spalte führen.

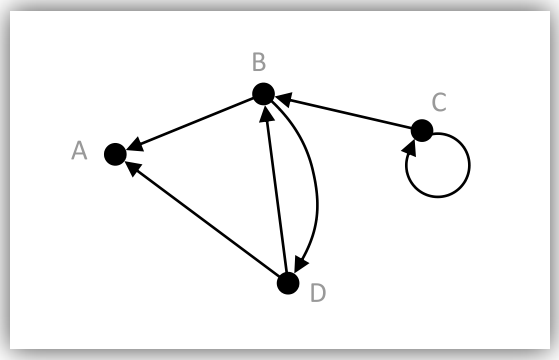


Abb. 35 Beispielgraph für die Speicherung

	A	B	C	D
A	0	0	0	0
B	1	0	0	1
C	0	1	1	0
D	1	1	0	0

Abb. 36 Adjazenzmatrix für den Graphen aus Abb. 35

Ein kleines Beispiel zeigen Abb. 35 und Abb. 36.

Die Vorteile der Adjazenzmatrix sind sowohl die Anschaulichkeit als auch die Anwendbarkeit einiger Algorithmen, wie z.B. dem Algorithmus von Warshall zur Ermittlung kürzester Wege. Ein gravierender Nachteil ist der unnötig große Speicherplatzverbrauch bei Graphen mit vielen Knoten und wenigen Kanten. Eine Navigation im Graph entlang von Kanten ist mit mittlerem Aufwand möglich, indem jeweils die Zeile eines Knotens für ausgehende oder die Spalte für eingehende Verbindungen durchlaufen wird. Das Erzeugen und Entfernen von Kanten ist sehr einfach, wohingegen der hohe Aufwand bei Einfügen oder Entfernen eines Knotens ein besonderer Nachteil der Adjazenzmatrix ist. Denn dazu muss die Größe der Matrix geändert werden.

Die zweite Möglichkeit der Speicherung eines Graphen ist die verkettete Speicherung. Dabei besitzt der Graph einen Verweis auf einen Startknoten und jeder Knoten (bis auf den letzten) besitzt einen Verweis auf einen nachfolgenden Knoten. Dadurch entsteht, ausgehend von einem Startknoten, eine einfach verkettete Knotenliste. Zusätzlich besitzt jeder Knoten einen Verweis auf eine Kantenliste. Diese Kantenliste existiert für jeden Knoten separat und deren Elemente enthalten Verweise auf andere Knoten.

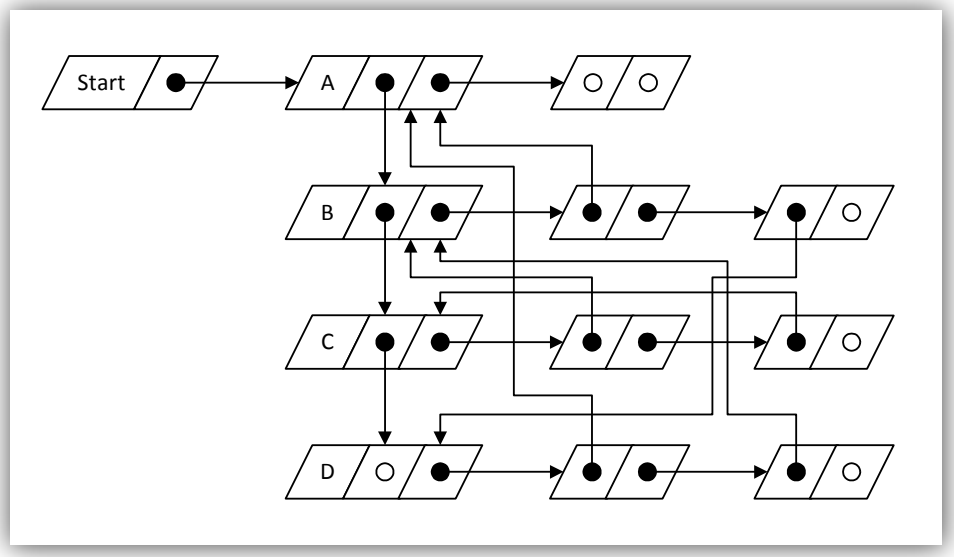


Abb. 37 Listenspeicherung des Graphen aus Abb. 35

Diese Variante ist zwar nicht besonders anschaulich, sie benötigt jedoch nicht mehr Speicherplatz als wirklich erforderlich und die Navigation im Graphen ist etwas schneller. Was der Nachteil der Adjazenzmatrix ist, ist der Vorteil der Listenspeicherung: Für das Einfügen und Löschen von Knoten wird nur sehr wenig Aufwand benötigt. Auch das Erzeugen und Entfernen von Kanten ist mit keinem hohen Aufwand verbunden.

Um zwischen den beiden Ansätzen zu unterscheiden, soll das voraussichtliche Zugriffsmuster der Laufzeitumgebung auf die Datenstruktur des Graphen prognostiziert werden.

Bei der sehr häufig auftretenden Weitergabe von Marken über Verbindungen, hat die Listenspeicherung den Vorteil, dass nicht alle Knoten (eine Zeile in der Adjazenzmatrix) nach einer bestehenden Verbindung durchsucht werden müssen. Der Nachbereich eines Knotens ist direkt in der Kantenliste erreichbar. In der Adjazenzmatrix kann der Nachbereich eines Knotens genauso leicht ermittelt werden wie der Vorbereich. Da die einfache Listenspeicherung die Kanten nur vorwärts verkettet, ist es mit ihrer Hilfe aufwändiger, den Vorbereich zu ermitteln. Falls ersichtlich wird, dass häufig eine Navigation in den Vorbereich eines Knotens nötig ist, müssen die Kanten bidirektional verkettet werden. D.h. dass jeder Knoten nicht nur die Nachfolger in einer Kantenliste speichern muss, sondern auch die Vorgänger. Dadurch wächst der Aufwand für die Änderung der Graphen-Struktur etwas an.

Geändert wird der Graph hauptsächlich während der Entwicklungsphase im Editor. Dabei werden sowohl Knoten eingefügt und entfernt als auch Verbindungen erzeugt und wieder gelöscht. Für derartige Strukturänderungen ist die Listenspeicherung generell besser geeignet als die Adjazenzmatrix.

Zur Erstellung, Bearbeitung und Ausführung des Graphen ist voraussichtlich kein besonderer Algorithmus notwendig, welcher von den Vorzügen der Adjazenzmatrix profitieren würde. Also wird als Ausgangspunkt für das Datenmodell des Graphen im Arbeitsspeicher die Listenspeicherung verwendet. Jedoch kann sie wahrscheinlich nicht in der klassischen Form verwendet werden, da die klassi-

sche Form keine Anschlüsse und auch keine rückwärtige Verkettung beinhaltet. Die genaue Form der Verkettung für DynamicNodes-Graphen wird weiter unten beschrieben.

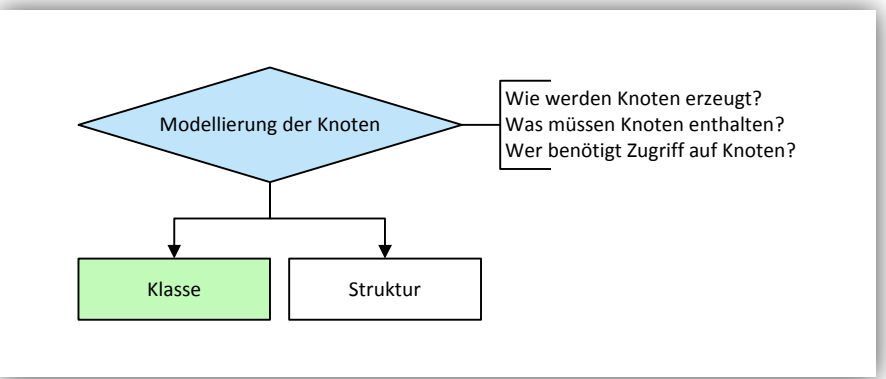


Abb. 38 Entwurfsentscheidungsmöglichekeiten für die Modellierung der Knoten

Als nächstes muss entschieden werden, in welcher Form ein Knoten im Arbeitsspeicher abgelegt wird. Im Kontext der .NET-Datentypen kommt dafür ein Wert- oder ein Verweistyp in Frage. Also kann der Knoten als Struktur oder als Klasse entworfen werden. Eine Struktur wird jedesmal kopiert, wenn sie einer Variablen zugewiesen wird. Das ist für einen Knoten nicht sinnvoll, da die Knoten nur einmal im Arbeitsspeicher vorliegen sollen, um Speicherplatz zu sparen und damit Änderungen an einem Knoten sofort für alle Leseoperationen auf diesem Knoten sichtbar sind.

Der Datentyp mit dessen Hilfe ein Knoten im Arbeitsspeicher repräsentiert wird, ist also ein Verweistyp. Erzeugt wird ein Knoten als Instanz einer Klasse, und ein Verweis auf diese Instanz wird in einer Knotenliste gespeichert, welche zur Datenstruktur des Graphen gehört. Damit wird der erste Teil der Listenspeicherung auf die OOP abgebildet.

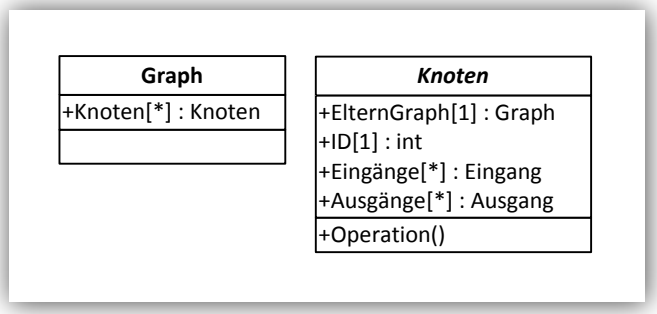


Abb. 39 UML-Diagramm von Graph und Knoten

Nun fehlt noch die Abbildung der Verbindungen. Nach der Spezifikation muss für einen Knoten eine feste Anzahl an Aus- und Eingängen definiert sein. Verbindungen beginnen und enden nicht direkt in einem Knoten, sondern in Ein- und Ausgängen.

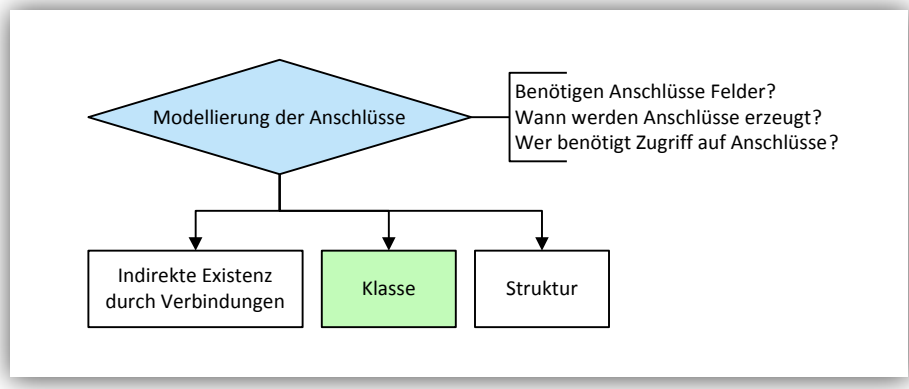


Abb. 40 Entwurfsentscheidungsmöglichkeiten für die Modellierung der Anschlüsse

Diese Anschlüsse werden vom Knoten während seiner Initialisierung erzeugt und müssen in der Datenstruktur des Knotens gespeichert werden. Doch zunächst muss geklärt werden, ob ein Anschluss überhaupt Speicherplatz benötigt. Da Anschlüsse eine Lesekopie oder sogar eine Queue enthalten, benötigen sie Speicherplatz und müssen von einem Datentyp repräsentiert werden.

Für die Anschlüsse gilt ähnliches wie für die Knoten: Es ist nicht sinnvoll, die Datenstruktur bei der Zuweisung zu einer Variablen zu kopieren. Änderungen (z.B. die Aktualisierung mit einer neuen Marke) müssen für alle lesenden Operationen sofort sichtbar sein. Demzufolge sollten Anschlüsse ebenfalls Verweistypen (Klassen) sein. Zu der Datenstruktur eines Knotens muss eine Liste mit Anschluss-Verweisen gehören. An dieser Stelle weicht der Entwurf von einer klassischen Listenspeicherung ab, denn die Klasse Listenspeicherung sieht keine Anschlüsse vor. Es ist sogar sinnvoll, für die Ein- und Ausgänge getrennte Listen zu verwenden, weil es häufiger vorkommt, dass eine Kategorie von Anschlüssen benötigt wird und nicht alle Anschlüsse.

Nun muss die Datenstruktur für Anschlüsse näher betrachtet werden. Es gibt zweifellos Unterschiede zwischen Ein- und Ausgängen. Es stellt sich jedoch die Frage, ob die Unterschiede sich auf die Datenstruktur auswirken und ob es auch Gemeinsamkeiten zwischen Ein- und Ausgängen gibt. Ein Unterschied, welcher sich auf die Datenstruktur auswirkt, ist z.B. dass ein Eingang eine Queue besitzt, ein Ausgang hingegen keine benötigt. Ein weiterer Unterschied ist, dass ein Eingang immer nur mit einem Ausgang verbunden sein kann, ein Ausgang hingegen mit mehreren Eingängen.

Gemeinsamkeiten sind z.B. die Lesekopie und dass Ein- und Ausgänge mit einer neuen Marke aktualisiert werden können. Auch sollte ein Anschluss immer einen Verweis auf seinen Knoten besitzen. Es gibt also sowohl Gemeinsamkeiten als auch Unterschiede zwischen Ein- und Ausgängen, welche sich beide auf die Datenstruktur auswirken.

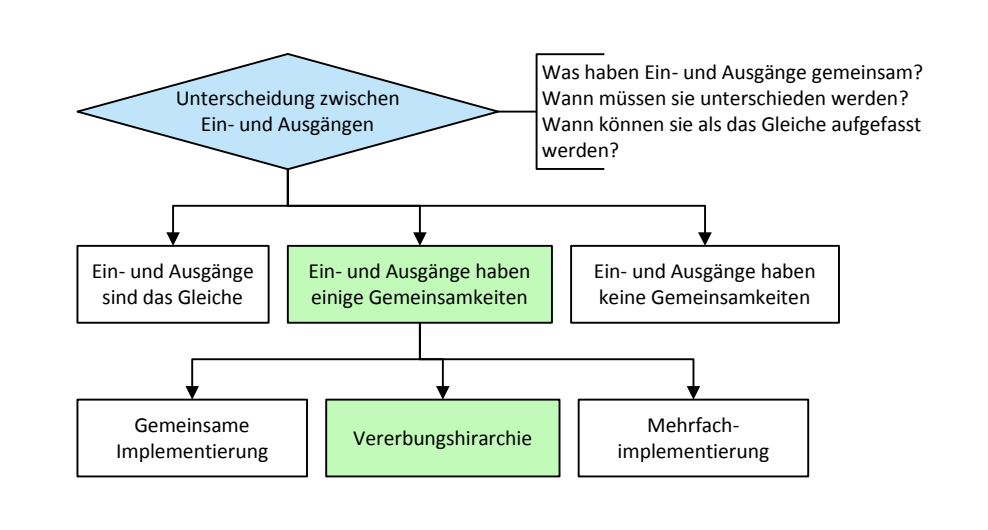


Abb. 41 Entwurfsentscheidungsmöglichkeiten für die Modellierung von Ein- und Ausgängen

Um diese Eigenschaften auf die Datenstruktur abzubilden, gibt es mehrere Möglichkeiten. Es könnte z.B. eine Klasse entworfen werden, welche die Fähigkeiten von Ein- und Ausgängen besitzt und Instanzen der Klasse nur mit einem Flag unterschieden werden. Dabei würde der Speicherplatz jedoch nicht effizient genutzt werden, weil z.B. ein Ausgang eine Queue besitzen würde, die nie verwendet wird.

Eine weitere Möglichkeit ist der Entwurf von zwei unabhängigen Klassen, die jeweils nur die nötigen Member für Ein- bzw. Ausgänge enthalten. Dabei müssten aber die Felder und Methoden, welche für Ein- und Ausgänge identisch sind, doppelt implementiert werden, was wiederum aus der Quellcode-Perspektive nicht effizient ist. Die beste Lösung ist die Nutzung von Vererbung. Dabei wird eine Basisklasse entworfen, welche alle Fähigkeiten implementiert die den Ein- und Ausgängen gemein ist. Zwei abgeleitete Klassen spezialisieren die Basisklasse anschließend für Ein- bzw. Ausgänge, indem sie die besonderen Fähigkeiten implementieren.

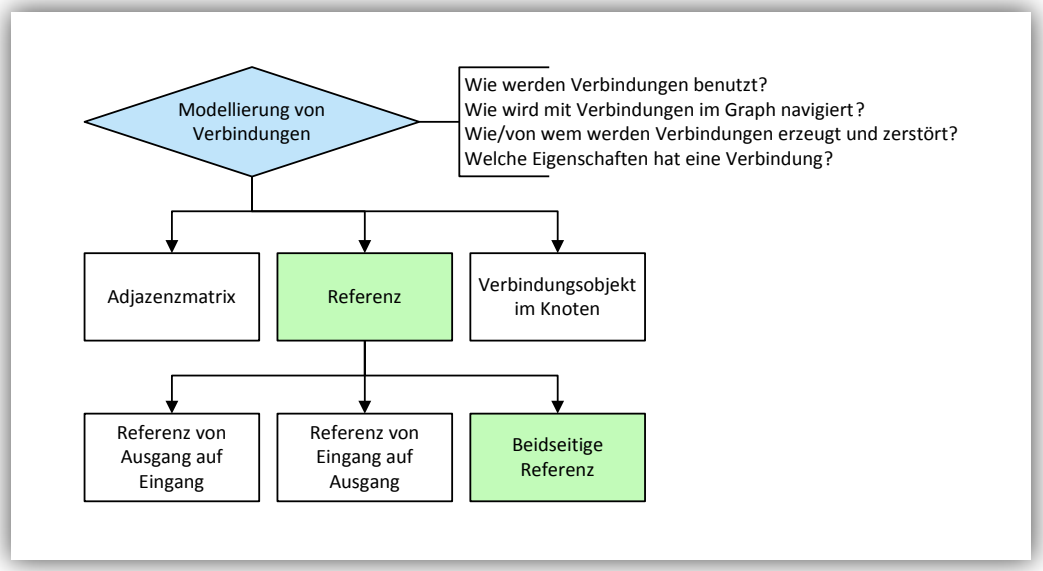


Abb. 42 Entwurfsentscheidungsmöglichkeiten für die Modellierung von Verbindungen

Um die Verbindungen im Datenmodell abzubilden, gibt es auch unterschiedliche Möglichkeiten. Die Adjazenzmatrix kommt nicht mehr in Frage, aber es ist zu entscheiden, ob die Verbindung eine eigene Datenstruktur benötigt oder ob sie durch einen oder mehrere Verweise repräsentiert werden kann.

Eine Verbindung besitzt theoretisch eine Kapazität, welche jedoch durch die Queue in der Datenstruktur der Eingänge repräsentiert wird. Dabei wird deutlich, dass zwischen den eigentlichen Verbindungen und den Eingängen eine (0..1:1)-Beziehung besteht. Deshalb können alle später evtl. erforderlichen Eigenschaften einer Verbindung in der Datenstruktur der Eingänge untergebracht werden. Die Verbindung selbst benötigt demzufolge keine eigene Datenstruktur. Es reichen Verweise (Referenzen) aus, um Verbindungen zwischen Ein- und Ausgängen zu repräsentieren.

Nun stellt sich die Frage, in welche Richtung die Verweise zeigen sollten. Ist eine reine Vorwärtsverkettung, wie in der klassischen Listenspeicherung angebracht ,oder gar eine reine Rückwärtsverkettung. Die Vorwärtsverkettung ist für die schnelle Weitergabe von Marken unverzichtbar, wie weiter oben bereits festgestellt wurde. Eine Rückwärtsverkettung ist sinnvoll, falls der Vorbereich eines Knoten häufig ermittelt werden muss. In 4.3.2 ist eine Bedingung für die Aktivierung von Knoten formuliert, welche den Vorbereich eines Knotens auf Aktivität überprüft. Das bedeutet, dass bei jeder Aktivierung eines Knotens sein Vorbereich ermittelt werden muss. Folglich ist die Rückwärtsverkettung unbedingt sinnvoll.

Die Verbindungen werden also beidseitig in den Anschlüssen gespeichert. Jeder Ausgang besitzt eine Liste mit Verweisen auf alle Eingänge, mit denen er verbunden ist. Jeder Eingang besitzt einen Verweis auf den Ausgang, mit dem er verbunden ist.

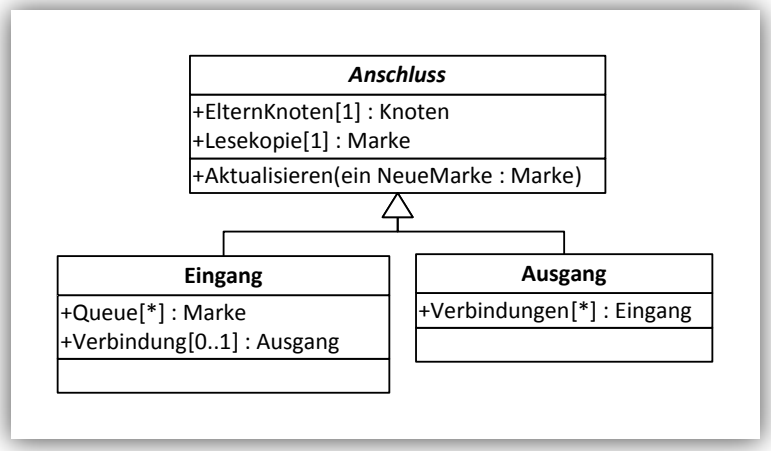


Abb. 43 Vererbungshierarchie von Anschlüssen

Auf der Basis der getroffenen Entscheidungen für das Datenmodell der Graphen-Struktur wird der Beispielgraph aus Abb. 44 in das Verweisnetzwerk in Abb. 45 überführt.

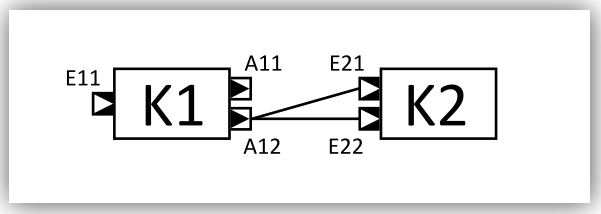


Abb. 44 DynamicNodes-Graph als Beispiel für Listenspeicherung

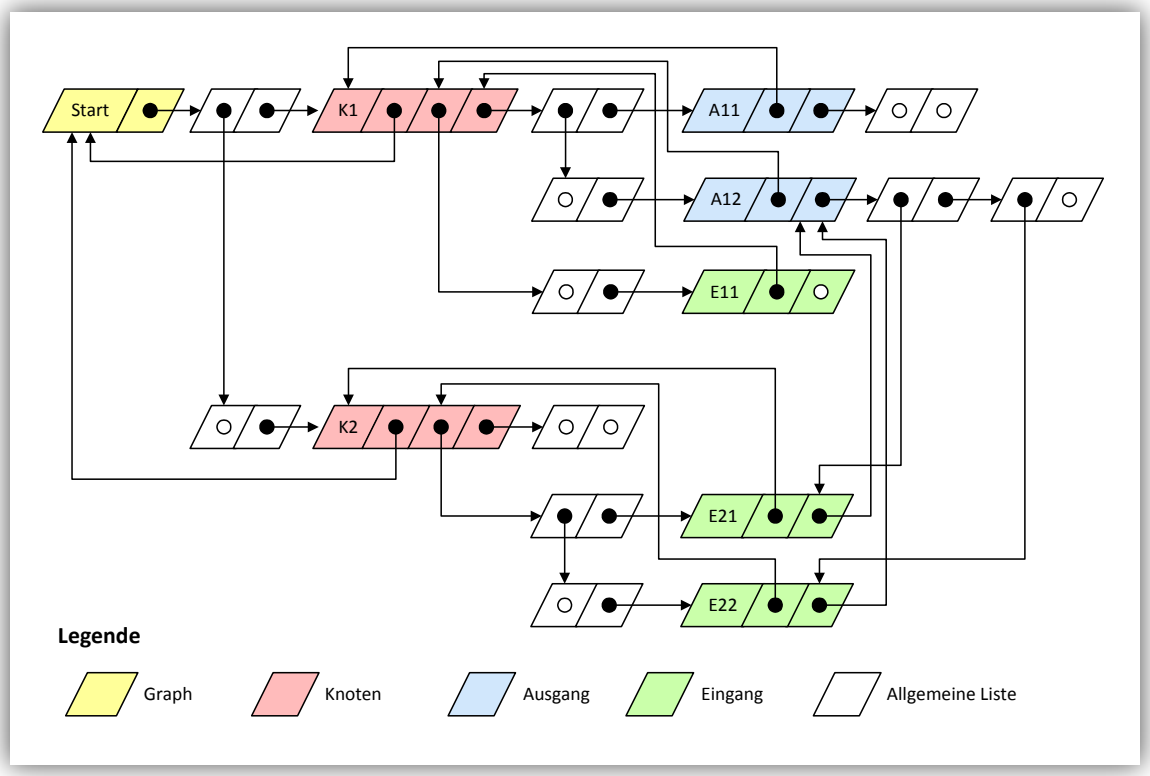


Abb. 45 Listenspeicherung des DynamicNodes-Graphen aus Abb. 44

Auffällig dabei ist, dass rückwärtige Verweise von Knoten auf den Graph und von Anschlüssen auf den Knoten existieren. Diese ermöglichen es z.B. für einen Knoten die Geschwisterknoten im Graph zu ermitteln. Oder es können für einen Eingang alle weiteren Anschlüsse des Knotens oder auch der Graph ermittelt werden.

Dem voraussichtlich zu erwartenden Zugriffsmuster auf die Datenstruktur des Graphen kann mit dem in Abb. 45 veranschaulichten Verweis-Konzept effektiv begegnet werden. Knoten können leicht eingefügt und entfernt werden, ohne dass die Datenstruktur des Graphen stark verändert werden muss. Es sind lediglich die bidirektionalen Verweise zu beachten, welche bei der klassischen Listenspeicherung nicht vorgesehen sind. Um Verbindungen zu erstellen oder zu löschen, muss die Verbindungsliste in dem Ausgang und der Verweis im Eingang manipuliert werden.

Die Beschaffung von neuem Speicherplatz für Knoten und Anschlüsse wird mit der Instanziierung von den oben skizzierten Verweistypen für Knoten und Anschlüsse durch die CLR erledigt. Beim Entfernen und Zerstören eines Knotens aus dem Graphen wird der Speicherplatz von Knoten und Anschlüssen durch den *Garbage Collector* der CLR wieder freigegeben.

Das UML-Diagramm in Abb. 46 veranschaulicht noch einmal die Assoziationen zwischen den Klassen des Datenmodells.

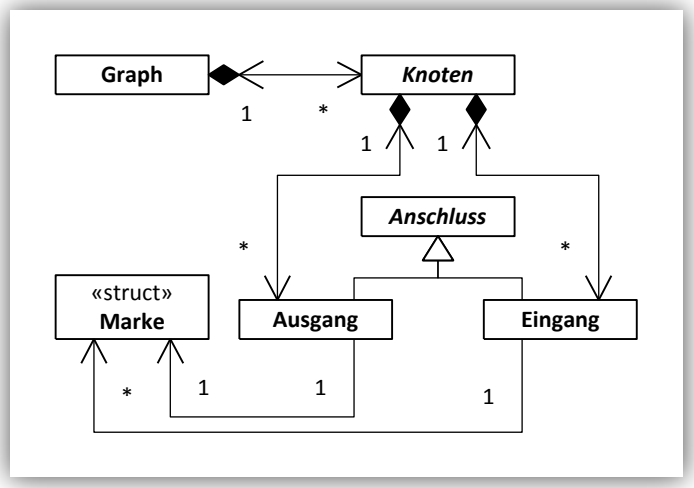


Abb. 46 UML-Diagramm des Datenmodells

4.4.3 Erweiterungen

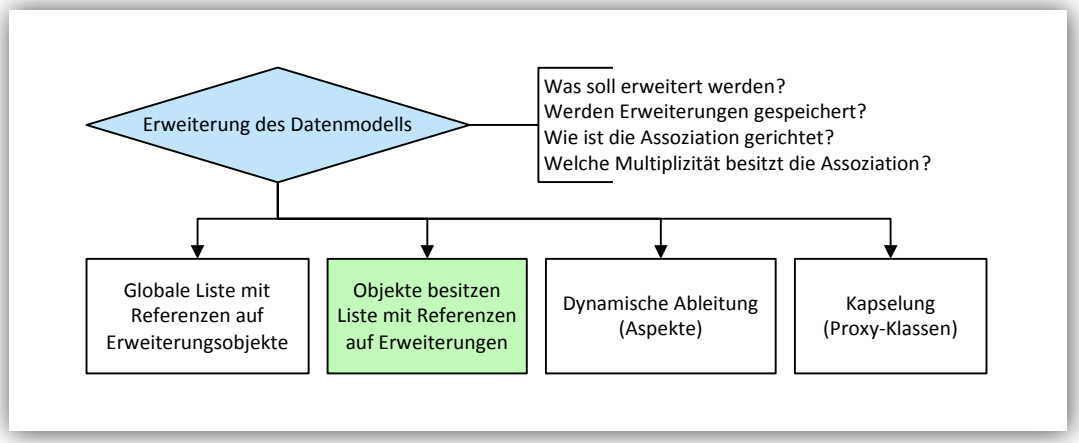


Abb. 47 Entwurfsentscheidungsmöglichkeiten für eine Erweiterung des Datenmodells

Die Spezifikation fordert in 3.5.1 eine Möglichkeit, die Basiselemente des Datenmodells (Graph, Knoten, Anschlüsse) um Objekte zu erweitern, welche noch nicht abzusehende Aufgaben erfüllen.
Das Datenmodell muss demzufolge eine (generische) Schnittstelle anbieten, welche möglichst unterschiedlichen Erweiterungen eine Anbindung bietet.

Die Erweiterungen sollen im Hinblick auf die Visualisierung des Graphen entworfen werden. Das Basisdatenmodell bietet an sich keine Unterstützung für visuelle Eigenschaften oder Methoden. Dieser scheinbare Mangel ist durchaus beabsichtigt, denn es ist vorstellbar, einen Graphen auf unterschiedliche Weise zu visualisieren. In dieser Arbeit ist der Entwurf einer 2D-Darstellung enthalten, jedoch soll der Entwurf des Datenmodells für alternative 2D-Darstellungen oder gar 3D-Darstellungen offen sein.

Um eine visuelle Darstellung von Graphen zu entwerfen, benötigt man zunächst Positionskoordinaten von Knoten, denn sie sollen nach der Spezifikation auf der Arbeitsfläche frei angeordnet werden können. Anschlüsse können eine relative Position zum Knoten besitzen. In der Datenstruktur von Eingängen werden die Eigenschaften von Verbindungen gespeichert. Dazu können im visuellen Bereich z.B. Umleitungspunkte einer Verbindung zählen, welche unübersichtliche Graphen anschaulicher machen können. Eine visuelle Erweiterung der Graphen-Datenstruktur könnte z.B. Methoden zum Darstellen des gesamten Graphen beinhalten.

Bei dieser Vorbetrachtung wird deutlich, dass es für jedes Element der Basisdatenstruktur sinnvolle Erweiterungen gibt. Und dass, obwohl nur die visuellen Erweiterungen betrachtet wurden. Es ist anzunehmen, dass weiterführende Erweiterungen ebenfalls an unterschiedlichster Stelle des Graphen (Graph, Knoten, Anschluss) anzubinden sein sollten.

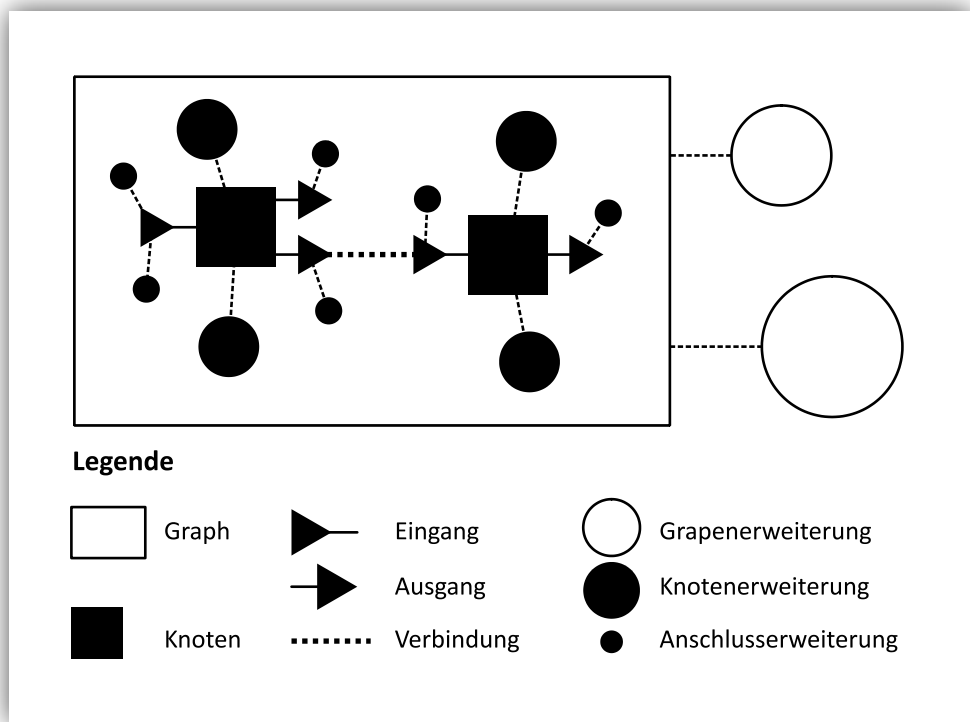


Abb. 48 Das Prinzip von Erweiterungen

Die Abb. 48 veranschaulicht die Assoziationen zwischen dem Basisdatenmodell und den Erweiterungen. Dabei werden eckige Formen für das Basisdatenmodell und runde für die Erweiterungen verwendet.

Um Erweiterungen mit dem Basisdatenmodell in Verbindung zu bringen, gibt es unterschiedliche Möglichkeiten. Die einfachste ist eine globale Liste mit Verweisen auf alle Erweiterungen in der Laufzeitumgebung. Dabei ist zusätzlicher Aufwand nötig, um einer Erweiterung mit einem konkreten Element eines Graphen (z.B. einem Knoten) in Verbindung zu bringen.

Eine weitere Möglichkeit ist, dass jedes Element des Basisdatenmodells eine Liste besitzt. Diese Liste kann Verweise auf alle Erweiterungen enthalten, welche sich auf das jeweilige Element bezieht. Z.B.

könnte in der Erweiterungsliste eines Knotens ein Verweis auf eine visuelle Knotenerweiterung enthalten sein, welche die grafischen Koordinaten des Knoten enthält.

Eine dritte Möglichkeit sind Aspekte. Aspekte sind ähnlich wie abstrakte Klassen. Sie können nicht instanziiert werden, sondern können einer Klasse zugewiesen werden. Jedoch sind Aspekte nicht an die vorgesehene Vererbungslinie gebunden, sondern können einer beliebigen Klasse zugewiesen werden. Sie werden z.B. in Enterprise-Anwendungen verwendet, um Protokollierungsfähigkeiten oder Sicherheitsfunktionen an beliebige Klassen anzufügen. Erweiterungen im Datenmodell könnten den Klassen des Basisdatenmodells als Aspekte zugewiesen werden. Für aspektorientierte Programmierung ist die .NET-Plattform jedoch von Hause aus nicht geeignet. Dazu wird eine zusätzliche Umgebung benötigt. Aspekte sollten nur eingesetzt werden, wenn sich ein Problem gar nicht oder nur sehr ineffektiv mit den objektorientierten Mitteln lösen lässt.

Man könnte als fünfte Variante auch Proxy-Klassen verwenden, welche von den Knoten erben und an deren Stelle instanziiert werden könnten. Jedoch ist dieses Vorgehen wegen den Knotenbibliotheken nicht möglich. Denn die Proxy-Klasse für die visuellen Fähigkeiten eines Knotens müssten für jeden Knotentyp extra implementiert werden, da sonst keine Vererbung möglich wäre.

Die zweite Möglichkeit ist die beste Lösung, um eine hohe Flexibilität mit einem übersichtlichen Design zu verbinden.

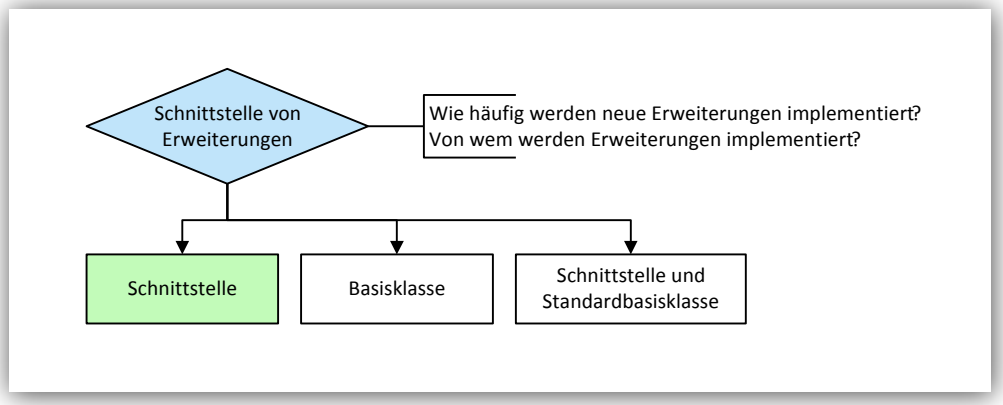


Abb. 49 Entwurfsentscheidungsmöglichkeiten für die Schnittstelle von Erweiterungen

Die Schnittstelle zwischen dem Basisdatenmodell und den Erweiterungen besteht aus zwei Seiten. Zunächst soll die Seite der Erweiterungen betrachtet werden. Jede Erweiterung sollte eine einheitliche Menge von Methoden und Eigenschaften besitzen, mit deren Hilfe eine Assoziation zu einem Objekt aufgebaut werden kann. Erweiterungen müssen einen Verweis auf das Objekt erhalten, dem sie zugewiesen werden. Erst dann können sie das Objekt um Fähigkeiten erweitern. Erweiterungen müssen auch informiert werden, wenn das erweiterte Objekt zerstört wird.

Für die Definition dieser Menge von Methoden und Eigenschaften gibt es drei Varianten. Es könnte eine Basisklasse für alle Erweiterungen implementiert werden oder es wird eine Schnittstelle implementiert, welche die Methoden und Eigenschaften definiert. Auch eine Kombination ist möglich. Jedoch schränkt die Verwendung einer Basisklasse die Möglichkeiten beim Entwurf von Erweiterun-

gen unnötig ein. Erweiterungen können Teil des Editors oder der Laufzeitumgebung sein oder sie werden von einem Knotentyp mitgebracht.

Die zweite Seite der Schnittstelle zwischen Objekten des Basisdatenmodells und Erweiterungen sind die zu erweiternden Objekte des Basisdatenmodells.

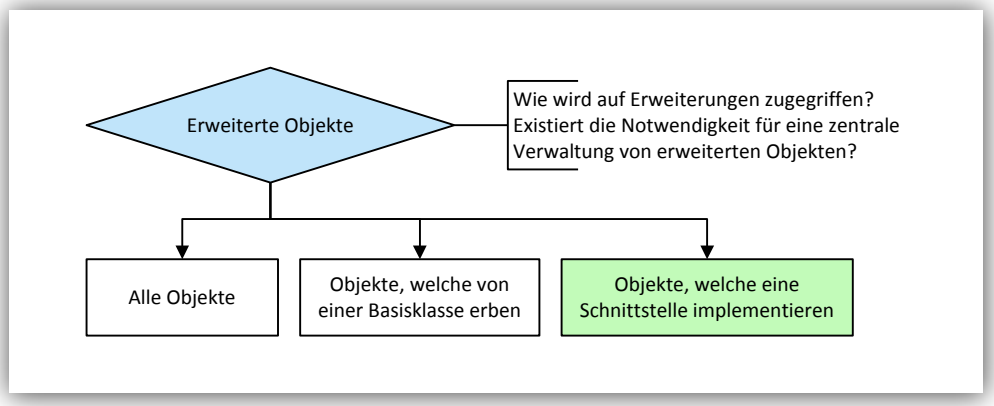


Abb. 50 Entwurfsentscheidungsmöglichkeiten für die Schnittstelle von erweiterbaren Objekten

Um Mechanismen für die Verwaltung von Erweiterungen zu erleichtern, ist es sinnvoll eine Schnittstelle zu definieren, welche die Erweiterungsfähigkeit von Objekten beschreibt. Denn genauso wie auf Seite der Erweiterungen sollten Objekte, welche erweitert werden sollen, eine Menge einheitlicher Methoden und Eigenschaften besitzen. Z.B. zum Aufbau bzw. Abbau von Bindungen zu Erweiterungen oder zur Abfrage angebundener Erweiterungen. Auch auf dieser Seite bietet sich die Definition durch eine Schnittstelle an, weil dadurch die Vererbungslinie der zu erweiternden Klassen nicht verändert werden muss.

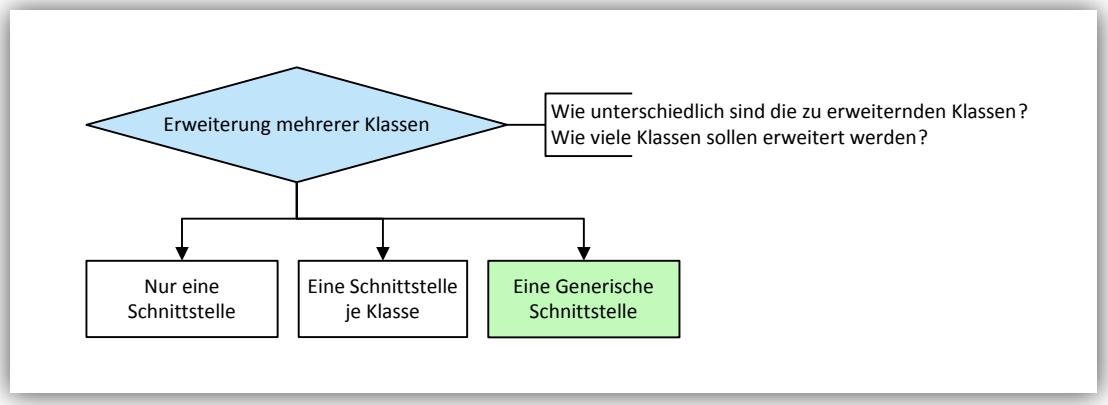


Abb. 51 Entwurfsentscheidungsmöglichkeiten für die Erweiterung mehrerer Klassen

Es sollen Graphen, Knoten und Anschlüsse eine Erweiterungsmöglichkeit bieten. Nun könnten alle drei bzw. vier Klassen (Ein- und Ausgänge getrennt betrachtet) eine Schnittstelle implementieren, welche die Assoziation mit einer Erweiterung vorsieht. Da die Erweiterungen dann alle mit derselben Schnittstelle kommunizieren würden, wäre es möglich z.B. eine visuelle Knoten-Erweiterung an einen Anschluss zu binden. An dieser Stelle ist mehr Typsicherheit wünschenswert. Dafür könnte für jede Klasse des Basisdatenmodells eine eigene Schnittstelle implementiert werden. Dieser Aufwand ist jedoch relativ hoch und der Aspekt der Erweiterbarkeit für Klassen wird im Code nicht gut abgebildet.

Eine generische Schnittstelle ist eine gute Lösung. Sie beschreibt in diesem Fall allgemein die Erweiterbarkeit einer Klasse, wird jedoch erst bei der Implementierung durch eine Klasse spezialisiert.

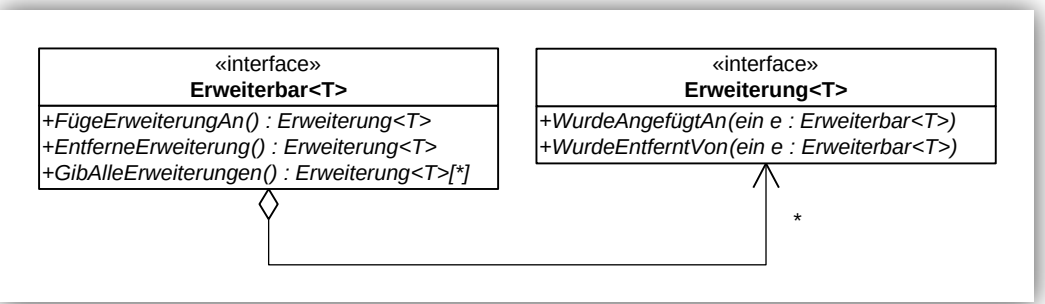


Abb. 52 Generische Schnittstellen für Erweiterungen und erweiterbare Klassen

Auf das Prinzip von generischen Schnittstellen, Klassen und Methoden soll an dieser Stelle nicht genauer eingegangen werden. Stattdessen soll auf ein Handbuch für C# und .NET 2.0 verwiesen werden (Schäpers, Huttary, & Bremes, 2002).

In Abb. 53 ist die konkrete Anwendung der generischen Schnittstellen für die Basisklassen des Datenmodells dargestellt. Jede Klasse des Basisdatenmodells implementiert die generische Schnittstelle `Erweiterbar<T>`, wobei der Typenparameter `T` durch die Klasse des Basisdatenmodells ersetzt wird. Durch die Implementierung der Schnittstelle `Erweiterbar<T>` entsteht eine Komposition mit den jeweiligen Spezialisierungen der generischen Schnittstelle `Erweiterung<T>`. Dabei kann ein erweiterbares Objekt beliebig viele Erweiterungen assoziieren. Ob und wie viele erweiterbare Objekte von einer Erweiterung assoziiert werden, ist nicht festgelegt.

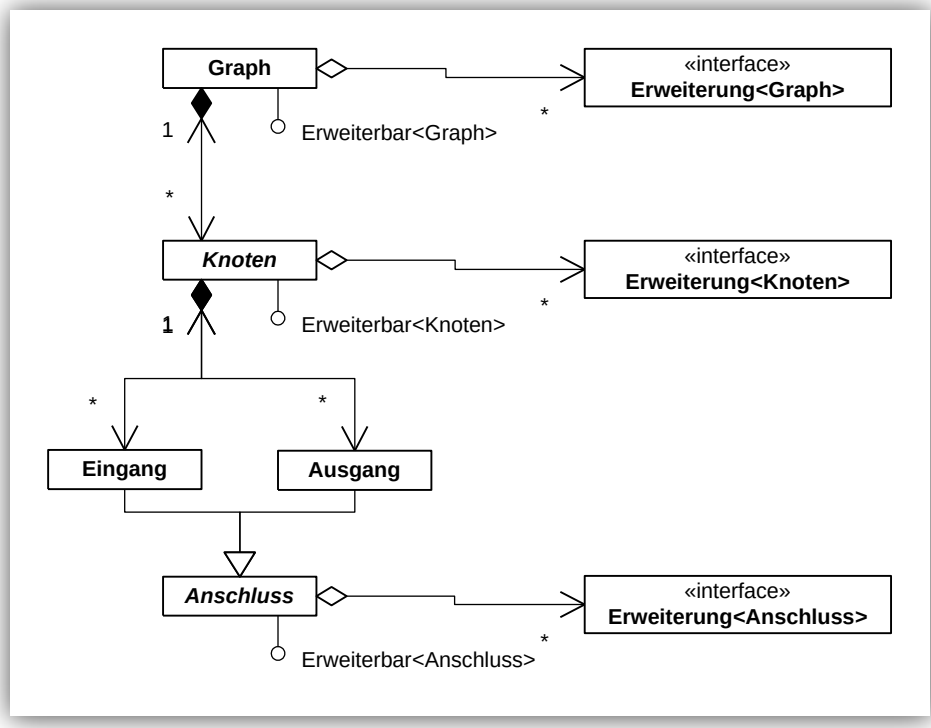


Abb. 53 UML-Diagramm des erweiterten Datenmodells

4.4.4 Speicherung in XML

Das Datenmodell eines DynamicNode-Programms wird nicht nur während der Ausführung im Arbeitsspeicher gehalten, sondern muss auch in Form einer XML-Datei auf der Festplatte abgelegt werden können. Dazu wird ein XML-Schema entworfen (Anhang G - Project.xsd; Band 2, S. 442), welches die Struktur jedes in XML persistierten DynamicNode-Programms definiert.

Um ein Programm in XML zu speichern, werden alle Eigenschaften wie Name und Beschreibung als XML-Tags abgelegt. Im Anschluss daran folgt eine Liste mit allen im Programm enthaltenen Knoten.

Für die Speicherung eines Knotens in XML wird zunächst lediglich die ID des Knotentyps benötigt. Zusätzlich wird seine Instanz-ID innerhalb des Programms gespeichert und er bekommt die Möglichkeit eine Anzahl benannter Eigenschaften in Form von Zeichenketten zu speichern, um evtl. einen internen Zustand zu sichern. Anschlüsse müssen prinzipiell nicht gespeichert werden, da sie vom Knotentyp definiert sind. Verbindungen müssen mit Startknoten, Startanschluss, Endknoten und Endanschluss gespeichert werden. Dabei dienen die Instanz-IDs der Knoten und die IDs der Anschlüsse als Referenz.

Alle Erweiterungen (von Programm, Knoten und Anschlüssen), die zum Zeitpunkt der Speicherung an ein Element des Datenmodells angefügt sind, bekommen die Möglichkeit eine benannte Liste mit Eigenschaften zu speichern. Diese Liste wird in Form von XML-Tags im Kontext des jeweiligen Elements gespeichert. Zu diesem Zweck wird auch für jeden Anschluss ein XML-Tag geschrieben und evtl. mit den Eigenschaften der angefügten Erweiterungen gefüllt.

Ein Beispiel für ein gespeichertes Programm findet sich in Anhang G - Alternative.dyn (Band 2, S. 451).

4.5 Abbildung der Parallelität

Operationsknoten und Programme sollen parallel ausgeführt werden können (vergl. 3.5.1). Dabei soll die Ausführung wenn möglich auf mehrere Prozessoren bzw. Prozessorkerne skalieren.

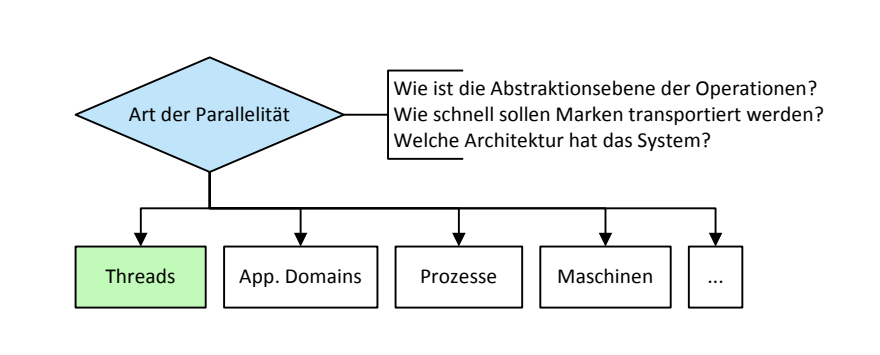


Abb. 54 Entwurfsentscheidungsmöglichkeiten für die Abbildung der Parallelität

Um die parallele oder pseudo parallele Ausführung von Maschinencode zu erreichen, gibt es unter modernen Betriebssystemen wie Windows 2000 oder Linux mindestens drei Möglichkeiten. Mit dem .NET-Framework kommt noch eine dritte hinzu. Die einfachste Möglichkeit ist natürlich den Maschinencode auf mehreren Maschinen (PCs) parallel auszuführen. Das ist jedoch für DynamicNodes nicht vorgesehen, da die Programmierumgebung auf einer einzelnen Maschine arbeiten soll (vergl. 4.1.4).

Die drei weiteren Möglichkeiten sind die Ausführung von Maschinencode in mehreren Prozessen auf dem Betriebssystem, die Ausführung von Maschinencode in mehreren Anwendungsdomänen innerhalb eines Prozesses und die Ausführung in mehreren Threads (Ausführungspfaden) innerhalb einer Anwendungsdomäne. Die .NET-Bibliothek stellt Mittel zur Verfügung, um Prozesse zu starten und Daten zwischen Prozessen auszutauschen (`System.Diagnostics.Process` und .NET-Remoting), Anwendungsdomänen innerhalb eines Prozesses zu verwalten und Daten zwischen ihnen auszutauschen (`System.AppDomain` und .NET-Remoting) und auch Threads zu erzeugen, diese zu synchronisieren und Daten zwischen ihnen auszutauschen (`System.Threading` und gemeinsamer Speicher).

Eine Anwendungsdomäne (Application Domain) ist im .NET-Framework eine Art Unteraufteilung des Prozessraums. Anwendungsdomänen haben keinen gemeinsamen Speicher, deshalb existieren z.B. statische Felder in jeder Anwendungsdomäne einmal. Wenn ein Assembly in eine Anwendungsdomäne geladen wurde, ist es nicht mehr möglich sie aus der Anwendungsdomäne zu entfernen. Es ist aber wohl möglich eine Anwendungsdomäne aus einem Prozess zu entfernen und damit auch die Assemblies zu entladen, welche von dieser Anwendungsdomäne geladen wurden.

Um hier eine Entscheidung zu treffen, muss der Kontext der Parallelität in DynamicNodes noch einmal kurz betrachtet werden. DynamicNodes muss in der Lage sein viele Programme parallel auszuführen. Also sind allein auf dieser Ebene mehr parallele Ausführungspfade zu erwarten als Prozessoren bzw. Prozessorkerne heutzutage in einem Arbeitsplatz-PC üblich sind. Ein Arbeitsplatz-PC besitzt üblicherweise 1 bis 4 Prozessoren bzw. Prozessorkerne. Bei einem Server sind dagegen schon bis zu 16 Kerne zu erwarten. Nimmt man nun an, dass in jedem der parallel auszuführenden DynamicNodes-Programme auch mehrere Knoten zu selben Zeit aktiv sind, wird deutlich, dass nicht jedem aktiven Knoten ein Prozessor(kern) zugeordnet werden kann.

Folglich ist hier neben echter Parallelität eine pseudo parallele Verarbeitung (z.B. mittels eines Zeitscheibenverfahrens) sinnvoll. Wenn unter .NET mehrere Prozesse, Anwendungsdomänen oder Threads verwendet werden, ist deren parallele Ausführung von dem darunterliegenden Betriebssystem abhängig. Windows verwaltet mindestens ab der Version 5.0 (Windows 2000) Prozesse und Threads automatisch echt und pseudo parallel. Für aktuelle Linux-Kernel gilt das in gleicher Weise. Folglich brauch beim Entwurf der Parallelität die abstrakte Ebene von Prozessen und Threads nicht verlassen zu werden, um eine automatische Skalierung auf mehrere Prozessoren bzw. Prozessorkerne zu erreichen.

In der Regel werden mehr Ausführungspfade aktiv sein als Prozessoren bzw. Prozessorkerne zur Verfügung stehen. Demzufolge wird das Betriebssystem einen mehr oder weniger häufigen Wechsel bei

der Zuweisung von Ausführungspfaden zu einem Prozessor(kern) vornehmen. Dieser Kontext-Switch genannte Wechsel erfordert einigen zeitlichen Aufwand. DynamicNodes soll derart entworfen werden, dass eine hohe Anzahl von Knoten effizient ausführbar ist, also wird auch bei einem Server mit 16 Prozessor(kern)en häufig ein Kontext-Switch notwendig sein, um eine gleichmäßige Ausführung aller aktiver Knoten zu erreichen.

Nun ist der Overhead bei einem Prozess-Kontext-Switch wesentlich größer als der bei einem Thread-Kontext-Switch. Der Kontext-Switch zwischen Anwendungsdomänen wird in etwa mit dem zwischen Threads übereinstimmen, da anzunehmen ist, dass Anwendungsdomänen von .NET auf Threads abgebildet werden. Also ist die Verwendung von Threads voraussichtlich günstiger.

Threads haben den Anwendungsdomänen gegenüber den weiteren Vorteil des gemeinsamen Speichers. Denn bei der Kommunikation zwischen Threads ist lediglich eine geschickte Synchronisation erforderlich, welche unter Umständen sehr performant arbeiten kann, insbesondere wenn solch klare Datenabhängigkeiten definiert sind, wie es durch die Verbindungen in den DynamicNodes-Graphen der Fall ist. Bei Anwendungsdomänen hingegen ist eine Kommunikation mit Hilfe von .NET-Remoting erforderlich. Dabei müssen Objekte, welche die Domänengrenze überqueren, serialisiert und deserialisiert werden. Um Objekte per Verweis zu übertragen, müssen Proxy-Objekte generiert werden (`System.MarshalByRefObject`), deren Methodenaufrufe wiederum Kommunikation über die Domänengrenze hinweg verursachen.

Also ist es am besten, die Parallelität der Knoten mit Hilfe von Threads abzubilden.

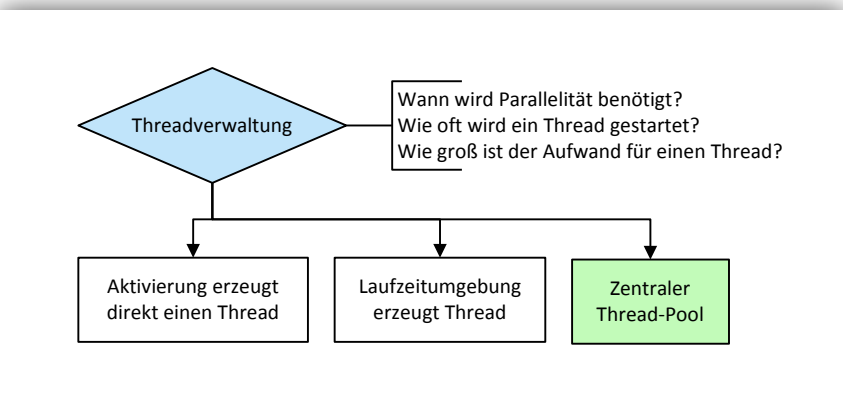


Abb. 55 Entwurfsentscheidungsmöglichkeiten für eine Threadverwaltung

Nun stellt sich die Frage, ob eine explizite Threadverwaltung notwendig ist und in welcher Form diese stattfinden sollte. Immer wenn ein Knoten aktiviert wird und er für die parallele Ausführung gekennzeichnet ist, wird ein Thread benötigt in dem die Arbeit des Knotens ausgeführt werden kann. Die einfachste Möglichkeit ist jedesmal einen Thread zu erzeugen, wenn einer benötigt wird. Dabei könnte die Laufzeitumgebung diese Aufgabe übernehmen, damit die Kontrolle für die Threaderzeugung an einer zentralen Stelle erfolgt.

Die zweite Möglichkeit ist die Verwaltung eines Thread-Pools. Dabei wird eine feste Anzahl von Threads erzeugt und bei Erzeugung sofort suspendiert. Wenn ein Thread benötigt wird, kann nun ein

Thread aus dem Pool aufgeweckt werden, damit er eine übergebene Aufgabe abarbeitet. Neben dem Pool mit den wartenden Threads gibt es bei diesem Konzept noch eine Queue mit anstehenden Aufgaben. Wenn alle Threads aus dem Pool aktiv sind, jedoch weitere Aufgaben mit einem Thread abgearbeitet werden sollen, dient diese Queue als Ablage für wartende Aufgaben. Wird ein Thread frei, erhält er sofort die nächste wartende Aufgabe. Ist die Queue leer, werden die Threads wieder suspendiert.

Das Konzept des Thread-Poolings hat den großen Vorteil, dass der Overhead zum Erzeugen eines Threads vermieden wird. Denn wenn bei jeder Knoten-Aktivierung ein Thread erzeugt und gestartet werden müsste, entstünde ein nicht unerheblicher Mehraufwand. Das Aufwecken eines Threads aus dem Thread-Pool hingegen ist mit keinem großen Aufwand verbunden. Demzufolge sollten Threads für die parallele Ausführung in DynamicNodes mit Hilfe eines Thread-Pools verwaltet werden.

4.6 Benutzeroberfläche

Dieses Kapitel beschreibt den Entwurf der Benutzeroberfläche. Unter der Benutzeroberfläche wird die Gesamtheit aller Fenster und Dialoge verstanden, welche die Interaktion zwischen Benutzer und DynamicNodes ermöglichen.

Die Spezifikation enthält in 3.5.2 die Anforderungen an den Editor. Dieser ist der Hauptteil der Benutzeroberfläche und enthält das Hilfesystem, wobei das Hilfesystem auch unabhängig vom Editor aufrufbar sein sollte. Das Testsystem hingegen ist ein separater Teil der Benutzeroberfläche.

Ein Steuerelement ist ein Bereich auf dem Bildschirm, der eine uni- oder bidirektionale Benutzerinteraktion für einen engen semantischen Kontext erlaubt. Beispiele für Steuerelemente sind z.B. ein Textfeld, eine Taste oder auch einen Schieberegler.

Unter dem Begriff Benutzerschnittstelle wird eine Gruppe von Steuerelementen verstanden, welche auf einem zusammenhängenden, meist rechteckigen Bereich des Bildschirms angezeigt werden und einen semantischen Bezug zueinander haben. Z.B. ist die Knotensteuerung genauso eine Benutzerschnittstelle wie die Steuerelemente zur Anzeige der Knotenbibliothek.

4.6.1 Ziele

Aus den funktionalen und nicht funktionalen Anforderungen der Spezifikation (vergl. 3.5.2, 3.5.5, 3.6.1) lassen sich einige allgemeine Entwurfsziele für die Benutzeroberfläche ableiten. Als Anhaltspunkte dienen dabei Stichworte wie: „intuitive Bedienung“, „selbsterklärend“, „ansprechende grafische Gestaltung“, „vorherrschende GUI-Gestaltung von Windows-Anwendungen“. So soll beim Entwurf besonders ***Benutzerfreundlichkeit***, ***Übersichtlichkeit*** und ***Flexibilität*** im Vordergrund stehen.

Benutzerfreundlichkeit bedeutet in diesem Zusammenhang, dass ein Benutzer, welcher das Programmiersystem nicht kennt, es jedoch gewohnt ist mit Windows-Anwendungen zu arbeiten, einen geringen Einarbeitungsaufwand benötigt.

Übersichtlichkeit bezieht sich hauptsächlich auf das Layout der Benutzeroberfläche. Dabei ist ein Layout übersichtlich, wenn ein Benutzer ohne langen Lernprozess erfassen kann, welche Art von Informationen die verschiedenen Bereiche der Oberfläche darstellen und eine große Menge an Informationen ohne Benutzerinterkation sichtbar ist.

Flexibilität des Layouts ist ein wichtiges Kriterium, weil es Knotentypen geben kann, welche Ein- und Ausgabebereiche in der Benutzeroberfläche benötigen („spezifische visuelle Ausgabebereiche“ in 3.5.2). Dadurch kann die Menge der Informationen, welche dargestellt werden müssen, abhängig von Knotentypen und Programm stark variieren.

Als Vorbilder für die GUI-Gestaltung dienen Benutzeroberflächen, welche ähnliche Anforderungen erfüllen müssen. Dazu zählen z.B. die IDE „Microsoft Visual Studio“ und das Framework „Eclipse“.

4.6.2 Beschriftungen

An dieser Stelle soll noch kurz ein Thema angesprochen werden, welches u.a. Grundlage für den Entwurf der Benutzeroberfläche ist. Es geht um konkrete Benennung und Beschriftung von Programmen, Knotentypen, Knoten und Anschlüssen.

Die Entwickler (von DynamicNodes selber und von Knotenbibliotheken) sollen die Möglichkeit haben Beschriftungen mehrsprachig anzulegen. Für den Benutzer hingegen soll es ausreichen, wenn er Beschriftungen in einer Sprache verfassen kann.

Ein Programm besitzt eine Zeichenkette als Name, welcher vom Benutzer frei gewählt werden kann. Ebenso besitzt ein Programm eine Zeichenkette als Beschreibung. Diese beiden Zeichenketten können nur in einer Sprache verfasst werden.

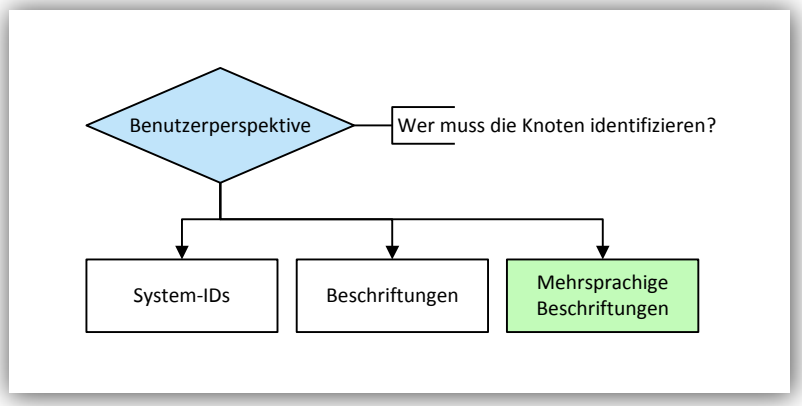


Abb. 56 Entwurfsentscheidungsmöglichkeiten für die Beschriftung von Knoten

Ein Knotentyp besitzt eine Zeichenkette als global eindeutige ID (z.B. der Klassenname), diese ist jedoch für die Benutzeroberfläche ungeeignet. Deswegen sieht die Spezifikation als Teil der Metadaten einen Namen und eine kurze Beschreibung in optional mehreren Sprachen vor. Der Name und die Kurzbeschreibung eines Knotentyps werden vom Knotenentwickler vorgegeben.

An dieser Stelle soll den Metadaten noch eine weitere Eigenschaft für Knotentypen hinzugefügt werden. Dabei handelt es sich ebenfalls um eine optional mehrsprachige Zeichenkette, nämlich um eine *Gruppe*. Damit kann der Knotenentwickler jeden Knotentyp einem Thema zuordnen. Z.B. kann ein Knoten für mathematische Grundrechenarten dem Thema „Mathematik“ zugeordnet werden. Die Gruppen können optional verschachtelt sein (z.B. „Mathematik/Grundrechenarten“).

Knoten (Instanzen) können ebenso wie Programme vom Benutzer mit einsprachigem Namen und Beschreibung versehen werden. Der Name einer Knoteninstanz sollte die Rolle des Knotens im Programm kennzeichnen.

Ein- und Ausgänge erhalten neben ihrer innerhalb des Knotens eindeutigen ID eine optional mehrsprachige Beschriftung. Ein Benutzer kann einen Anschluss anhand der Beschriftung, die Laufzeitumgebung anhand der ID identifizieren.

Ein wichtiges Thema sind in diesem Kontext noch die Sub-Graphen. Damit sind Graphen gemeint, welche als Operation in einen speziellen Knoten geladen werden. Dabei werden Super-Anschlüsse verwendet, um die Anschlüsse des Aggregationsknotens zu generieren (vergl. 2.1.3). Da Super-Anschlüsse vom Benutzer und nicht vom Knotenentwickler definiert werden, muss nicht unbedingt eine Möglichkeit bestehen Super-Anschlüsse mehrsprachig zu beschriften. Also wird als Beschriftung für Super-Anschlüsse die vom Benutzer definierte ID verwendet.

4.6.3 Inhaltliche und grafische Aufteilung

Die Aufteilung der Benutzeroberfläche geschieht einmal inhaltlich und einmal grafisch. Begonnen wird mit der inhaltlichen Aufteilung, eine grafische Aufteilung der Benutzerschnittstellen (Editor, Hilfe, Testumgebung) in Fenster und Bereiche wird anschließend vorgenommen.

Inhaltliche Aufteilung

Zunächst sollen die verschiedenen Aufgaben der Benutzeroberfläche gruppiert betrachtet werden. Dabei wird zusammengefasst was eine enge Beziehung hat. Die Gruppierung ist die Grundlage für den Entwurf der unterschiedlichen Bildschirm- bzw. Fensterbereiche:

- Programmverwaltung
  Öffnen, Speichern, Schließen, MDI10, Anzeigen
- Bearbeitung der Programmstruktur
  Einfügen und Entfernen von Knoten, Erzeugen und Zerstören von Verbindungen, grafisches Anordnen der Knoten auf der Arbeitsfläche
- Steuerung der Knoten
  Anzeigen der spezifischen Benutzerschnittstelle zur Konfiguration eines Knotens, Anzeigen der Markierung an Ein- und Ausgängen, Änderung von Marken an offenen Eingängen

[10] Multi Document Interface - beschreibt eine Anwendung, in der zur gleichen Zeit mehrere Dokumente geöffnet sein können

- Anzeigen von knotenspezifischen Ausgabebereichen
  Verwaltung einer Vielzahl von unterschiedlichen Benutzerschnittstellen, welche von Knoten erzeugt und zerstört werden können
- Anzeigen der Knotenbibliothek
- Konfiguration der Laufzeitumgebung
  Bereitstellung einer Benutzerschnittstelle zur Anzeige und Änderung unterschiedlicher Parameter der Laufzeitumgebung
- Hilfesystem
  Anzeigen einer Übersicht über die Inhalte der Hilfe und der Knotenreferenz, Anzeigen der Hilfetexte und der Knotenbeschreibungen
- Benutzerschnittstelle der Testumgebung
  Steuerelemente für das Testen von Knotentypen, Anzeigen der Testergebnisse

Grafische Aufteilung

Zunächst soll davon ausgegangen werden, dass der Editor in einem Fenster zusammengefasst wird. Ebenso soll die Benutzerschnittstelle der Testumgebung in einem Fenster zusammengefasst werden. Das Hilfesystem kann Teil des Editors sein oder aber in einem eigenen Fenster angezeigt werden.

Abb. 57 Beispiel für eine typische Windows-Nachrichten-Box

Jedes dieser Fenster hat die Möglichkeit Dialoge anzuzeigen. Dialoge sind modale Unterfenster, das bedeutet, dass das Hauptfenster deaktiviert ist, solange der Dialog angezeigt wird. Ein einfaches Beispiel für einen Dialog ist eine Windows-Nachrichten-Box wie in Abb. 57.

Hilfesystem

Am einfachsten aufzuteilen ist wohl das Hilfesystem. Die Benutzerschnittstelle muss für das Hilfesystem nur zwei Aufgaben erledigen. Dazu wird die Benutzerschnittstelle in zwei Bereiche geteilt. Die meisten Hilfesysteme besitzen links einen Bereich mit Inhalt, Index und Suche. Bei der Hilfe von DynamicNodes soll zunächst nur eine Inhaltsübersicht enthalten sein. Diese wird dann auch an den linken Rand platziert, damit der Benutzer seine Gewohnheit nicht ändern muss.

Der Inhalt des Hilfesystems besteht aus allgemeinen Hilfetexten und der Knotenreferenz. Die Knotenreferenz ist so wie die Knoten selber in einer Hierarchie strukturiert. Dabei dienen sowohl die Knotenbibliotheken als auch die thematischen Gruppen, welche den Knoten zugeordnet sind, als Hierarchie.

Die rechte Seite ist der Bereich in dem die Hilfetexte und die Knotenbeschreibungen angezeigt werden.

Abb. 58 Aufteilung für die Benutzerschnittstelle des Hilfesystems

Testumgebung

Die Testumgebung wird in drei Bereiche geteilt. Am linken Rand soll sich eine Liste mit allen Knotentypen befinden. An deren Fuß werden Tasten zum Starten der Testfälle, für einen in der Liste ausgewählten oder für alle Knotentypen, angeordnet. Der Hauptbereich wird horizontal geteilt. Der obere Bereich bietet Platz für die Ausgabe von visuellen Testfällen. Der untere Bereich ist eine Liste mit den Ergebnissen der Testfälle.

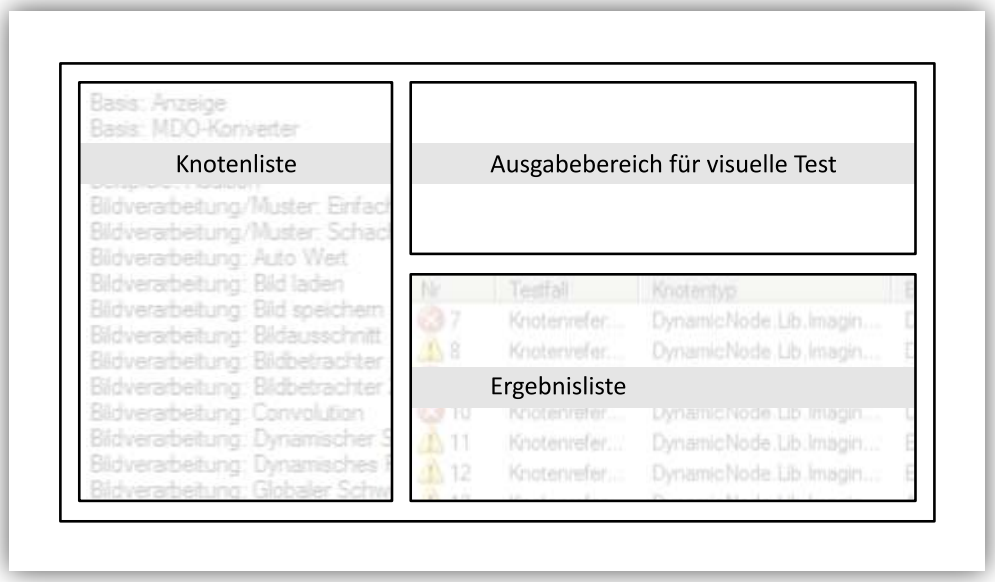


Abb. 59 Aufteilung für die Benutzerschnittstelle der Testumgebung

Editor

Der Editor ist die echte Herausforderung. Er muss eine Vielzahl von verschiedenen Benutzerschnittstellen in einer übersichtlichen Weise darstellen. Dennoch lässt sich eine grundlegende Aufteilung recht schnell ermitteln.

Verwaltungsaufgaben und MDI-Funktionen können in einem Menü und in einer Werkzeugleiste zweckmäßig untergebracht werden. Der Arbeitsbereich, auf dem die Programmstruktur mit Maus-interaktionen bearbeitet werden kann, sollte an zentraler Stelle im Editor Platz finden und nicht zu klein bemessen sein. Die Knotenbibliothek könnte sowohl links als auch rechts vom Arbeitsbereich angeordnet werden. Die Kontensteuerung und die Liste mit den Ein- und Ausgängen des markierten Knotens sollten rechts vom Arbeitsbereich zu finden sein, da dies Benutzerschnittstellen sind, welche kontextsensitiv sind. Kontextsensitiv bedeutet, dass ihr Inhalt sich ändert, sobald ein anderer Knoten ausgewählt wird.

Die Knotenbibliothek auch rechts anzuordnen wäre keine praktische Aufteilung. Der Arbeitsfluss ist in der Regel von links nach rechts, was der natürlichen Arbeitsrichtung eines rechtshändigen Mitteleuropäers entspricht. Links werden neue Knoten in der Knotenbibliothek ausgewählt, in der Mitte angeordnet und verbunden und rechts werden ihre Eigenschaften bearbeitet.

Wenn das Hilfesystem eingeblendet werden soll, kann es unter der Arbeitsfläche angezeigt werden, denn die Hilfetexte erfordern zum Lesen einen mehr horizontal als vertikal ausgerichteten Bereich.

Eigenschaften von Programmen, Knoten und Anschlüssen, welche nicht so häufig benötigt werden, sollten in Dialogen untergebracht werden, damit das Editor-Fenster nicht überladen wird.

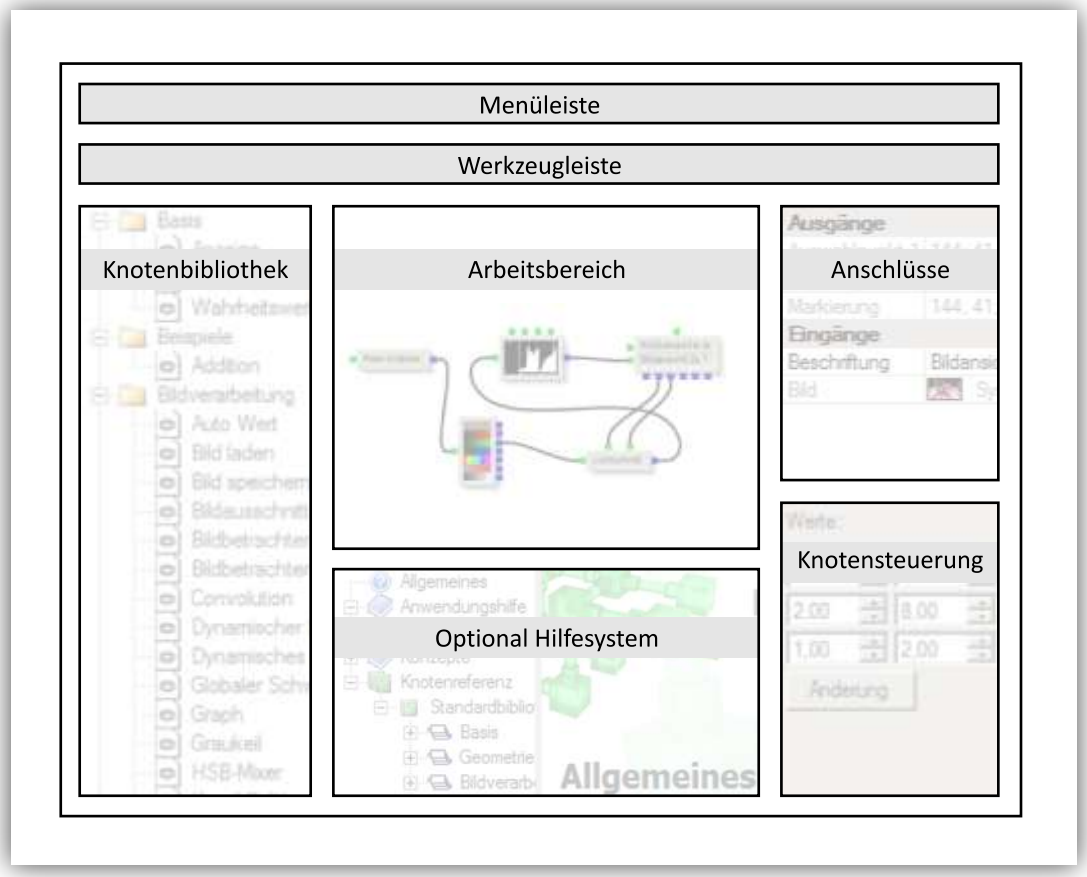


Abb. 60 Aufteilung des Editors

4.6.4 Arbeitsbereich

Der Arbeitsbereich ist die visuelle Darstellung eines Programms auf einer zweidimensionalen Ebene. Im Arbeitsbereich eines Programms werden die Knoten und Verbindungen als Boxen und Verbindungslinien dargestellt.

Die Spezifikation sieht vor, dass Knoten ihre Ausgabe auf dem Arbeitsbereich mitbestimmen können. Um das zu ermöglichen und dennoch einen homogenen optischen Eindruck bei vielen verschiedenen Knotentypen zu erreichen, wird der Ausgabebereich eines Knotens auf dem Arbeitsbereich in einen Rahmen und einen benutzerdefinierten (vom Knoten gezeichneten) Bereich aufgeteilt. Ein Knoten kann für sich definieren, ob sein Name oder sein Knotentyp im Rahmen angezeigt werden soll. Macht der Knoten keine Vorgaben für seine Darstellung, wird kein benutzerdefinierter Bereich gezeichnet und der Knotentyp angezeigt.

Ein- und Ausgänge werden als kleine Kästchen dargestellt, wobei sie sich durch unterschiedliche Farben unterscheiden sollten. Anschlüsse können an allen vier Seiten des Rahmens angeordnet werden. Bei der Implementierung der ersten Prototypen wurde deutlich, dass viele Knotentypen Ein- und Ausgänge für die zu verarbeitenden Daten und zusätzlich Eingänge für Daten besitzen, welche den Knoten für seine Aufgabe konfigurieren. Z.B. hat ein Knoten für das Weichzeichnen eines Bildes einen Ein- und Ausgang für das Quell- und Ergebnisbild und zusätzlich einen Eingang für die Stärke der Weichzeichnung. Um die Verbindungen in einem Programm übersichtlicher zu machen, hat sich bewehrt die „Daten-Anschlüsse“ links und rechts und die „Konfigurationsanschlüsse“ oben am Knotenrahmen zu platzieren. DynamicNodes kann die Anschlüsse jedoch nicht automatisch unterscheiden, demzufolge ist es Aufgabe des Knotenentwicklers die Position der Anschlüsse festzulegen. Die oben beschriebene Vorgehensweise ist als Empfehlung zu betrachten.

Macht der Knotenentwickler keine Vorgaben, werden Eingänge automatisch links und Ausgänge automatisch rechts angeordnet.

Verbindungen werden als Linien dargestellt. Eine optische Kennzeichnung der Flussrichtung einer Verbindung ist nicht nötig, da die unterschiedlichen Farben der Anschlüsse die Richtung leicht erkennen lassen.

Für die Erstellung und Bearbeitung von Programmen sollte eine Eingabetechnik verwendet werden, welche einem Windows-Benutzer bereits gut bekannt ist. Folgende Operationen machen den Hauptteil der Programmbearbeitung aus: Neue Knoten aus der Knotenbibliothek auf dem Arbeitsbereich platzieren, Knoten verschieben, Verbindungen erstellen, Verbindungen zerstören. Für alle diese Operationen bietet sich die Drag&Drop-Technik an.

Um eine Instanz eines Knotentyps zu erstellen und ihn in das Programm einzufügen, drückt der Benutzer die linke Maustaste, während sich der Cursor über dem entsprechenden Eintrag in der Knotenbibliothek befindet. Anschließend bewegt er den Cursor auf den Arbeitsbereich und lässt die Maustaste wieder los. Als Folge wird ein Knoten instanziiert, in das Programm eingefügt und folglich

auf dem Arbeitsbereich angezeigt. Dabei bekommt der Knoten die Koordinaten zugewiesen, die der Cursor auf dem Arbeitsbereich beim Loslassen der Maustaste besaß.

Um einen Knoten auf dem Arbeitsbereich zu verschieben, drückt der Benutzer die linke Maustaste während sich der Cursor innerhalb des Knotenrahmens des zu verschiebenden Knotens befindet. Dann bewegt er den Cursor an die Zielposition, an welcher der Knoten platziert werden soll. Dabei wird der Arbeitsbereich bei jeder Mausbewegung neu gezeichnet und der Knoten bewegt sich unter dem Cursor mit. Wenn die Maustaste losgelassen wird, bekommt der Knoten endgültig die neuen Koordinaten zugewiesen.

Um eine Verbindung zu erstellen, drückt der Benutzer die linke Maustaste während sich der Cursor über einem Ein- oder Ausgang befindet. Dann bewegt er den Cursor über den zu verbindenden Aus- oder Eingang. Während er den Cursor über die Arbeitsfläche bewegt, wird eine vorläufige Verbindungslinie vom Startanschluss zum Cursor gezeichnet und es werden alle Anschlüsse farblich gekennzeichnet zu denen eine gültige Verbindung aufgebaut werden kann. Lässt der Benutzer die Maustaste los, wenn der Cursor sich über keinem oder einem inkompatiblen Anschluss befindet, verschwindet die vorläufige Verbindungslinie und die Verbindungsoperation wird abgebrochen. Lässt der Benutzer die Maustaste hingegen los, während sich der Cursor über einem gültigen Zielanschluss befindet, wird eine Verbindung zwischen dem Startanschluss und dem Zielanschluss erzeugt und die Verbindungslinie wird permanent angezeigt.

Wenn der Startanschluss ein Eingang ist, zu dem bereits eine Verbindung existiert, wird diese automatisch zerstört, damit nicht mehr als eine Verbindung zu einem Eingang besteht.

Um eine Verbindung zu zerstören, genügt ein Mausklick auf den Eingang, in dem die Verbindung endet.

Ein Knoten kann mit einem Mausklick innerhalb seines Knotenrahmens markiert werden. Für markierte Knoten werden die kontextsensitiven Benutzerschnittstellen (Anschlüsse, Knotenkonfiguration) angezeigt. Ein markierter Knoten wird durch eine farbige Linie im Knotenrahmen gekennzeichnet.

4.6.5 Ansichten

In der bisherigen Aufteilung des Editors sind die Knoten-spezifischen Ausgabebereiche noch nicht berücksichtigt worden. Sie stellen das größte Problem dar, weil über ihre Anzahl und ihren Platzbedarf keine endgültige Annahme gemacht werden kann. Diese Parameter sind abhängig von der Anzahl der Knoten in den Programmen und der Nutzung der Ausgabebereiche durch die Knotenentwickler. Darüber hinaus hat sich während der Entwicklung des ersten Prototypen gezeigt, dass es sinnvoll, ist solche Ausgabebereiche außerhalb des Editor-Fensters, z.B. auf einem zweiten Bildschirm, anzuzeigen.

Um diesem Problem zu begegnen, ist die Verwendung von Dock-Fenstern sinnvoll. Das Prinzip der Dock-Fenster ist wahrscheinlich aus Microsoft Visual Studio bekannt. Eine große Anzahl von Unterfenstern können mittels Drag&Drop innerhalb oder außerhalb des Hauptfensters flexibel angeordnet und mit Karteireitern gestapelt werden.

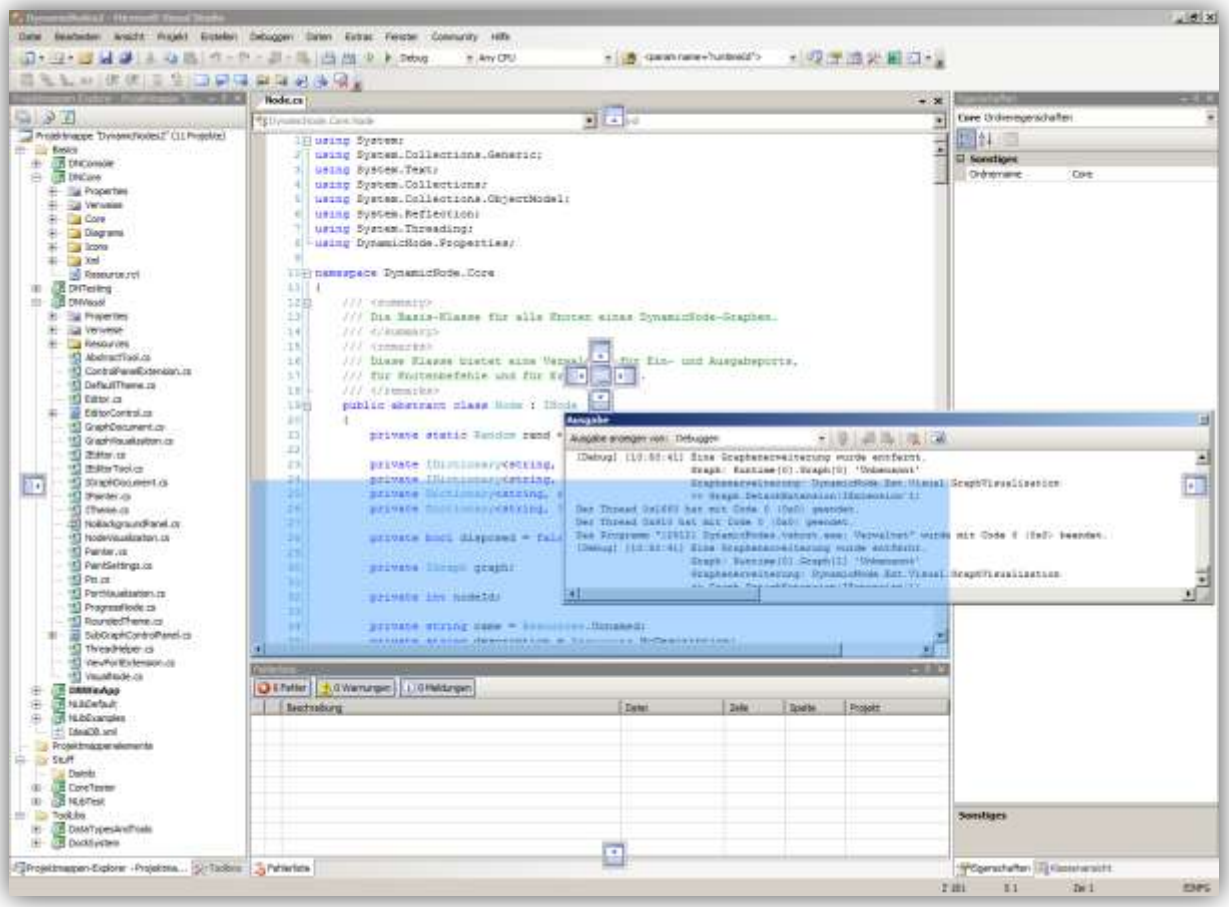


Abb. 61 Screenshot von Visual Studio während des Verschiebens eines Unterfensters

Ein solches flexibles Unterfenster wird oft auch Ansicht genannt. Wenn die verschiedenen Benutzerschnittstellen des Editors als Ansichten flexibel angeordnet werden könnten, würde das die Benutzerfreundlichkeit der Benutzeroberfläche von DynamicNodes stark verbessern und die oben genannten Probleme mit den knotenspezifischen Ausgabebereichen lösen.

Eine kurze Recherche hat ergeben, dass Microsoft die Steuerelementbibliothek für seine Dock-Fenster nicht mit den Entwicklungswerkzeugen mitliefert. Es gibt eine Vielzahl von Anbietern auf dem Markt, die Steuerelementbibliotheken als Pakete verkaufen, welche dann u.a. auch eine Form von Dock-Fenstern beinhalten. Jedoch belaufen sich die Preise für eine solche Steuerelement-Bibliothek zwischen 250 und 800 US-Dollar und bringen viel mehr mit als für DynamicNodes nötig. Genannt seien an dieser Stelle Anbieter wie BCG-Soft (www.bcgsoft.com) und Olivio IT Inc. (www.olivio.com).

Auf SourceForge fand sich dann ein Open Source Projekt mit dem Namen „DockPanel Suite“ von Weifen Lou, welches die Visual Studio Dock-Fenster nachbildet. Jedoch ließ sich mit dieser Steuerelemente-Bibliothek ein Problem nicht lösen: Wenn mehrere Programme in DynamicNodes

nacheinander geöffnet werden und Knoten in jedem Programm neue knotenspezifische Ausgabebereiche öffnen, kann deren Position nachträglich nicht rekonstruiert werden. Denn die „DockPanel Suite“ unterstützt zwar eine XML-Serialisierung des Layouts, diese kann jedoch nicht für einzelne Dock-Fenster verwendet werden.

Um eine Lösung für das Problem zu finden, wurde eine eigene prototypische Bibliothek für Dock-Fenster mit dem Namen „DockSystem“ implementiert. Dabei wurde deutlich, dass es keine triviale Lösung für die nachträgliche Integration von Fenstern in ein potentiell verändertes Layout gibt. Aus Zeitgründen wurde an dieser Stelle auf eine Weiterentwicklung des Dock-Fenster-Systems verzichtet.

Da aber deutliches Verbesserungspotenzial auf der Grundlage des Dock-Fenster-Prinzips erkennbar ist, sollen die Benutzerschnittstellen des Editors als Ansichten mit Hilfe der eigenen „DockSystem“-Bibliothek implementiert werden. Dadurch ist die flexible Aufteilung des Editors durch den Benutzer möglich und knotenspezifische Ausgabebereiche können frei platziert werden.

4.6.6 Eigenschaftsdialoge

Alle Steuerelemente werden für Parameter und Eigenschaften, welche nicht so häufig manipuliert werden müssen in Dialoge ausgegliedert. Das .NET-Framework bringt ein Steuerelement mit, welches alle öffentlichen Eigenschaften eines Objektes in einer Liste anzeigt und unter Umständen deren Änderung ermöglicht. Dieses Steuerelement heißt „PropertyGrid“. Damit wird die Gestaltung von vielen unterschiedlichen Eigenschafts- und Konfigurationsdialogen überflüssig.

Mit Hilfe der Flexibilität dieses Steuerelements ist es möglich einen einzigen Eigenschaftsdialog für viele verschiedene Objekte zu verwenden. Dem Dialog wird vor Anzeige das zu bearbeitende Objekt übergeben und dieser zeigt dann automatisch die verfügbaren Eigenschaften an, also für ein Programm z.B. den Programmnamen und die Beschreibung und für einen Anschluss ob er ein Super-Anschluss ist und welche ID er besitzt.

Für den Benutzer ist die Benutzeroberfläche leichter verständlich, wenn er einen Eigenschaftsdialog für viele verschiedene Objekte verwenden kann.

4.6.7 Look & Feel

Ein wichtiges Entwurfsziel für die Benutzeroberfläche ist, dass der Benutzer Spaß beim Umgang mit ihr hat. Um das zu erreichen, sind noch einige ästhetische Aspekte zu betrachten.

Die Farben der Benutzeroberfläche sollen sich nach den Farben des Betriebssystems richten. Auch bei der Visualisierung der Programme sollen Farben aus der Farbpalette des Betriebssystems verwendet werden. So dass z.B. ein Knoten die Farbe eines Dialogs oder einer Schaltfläche erhält. Die Rahmen der Knoten sind in dem Windows-typischen 3D-Look zu zeichnen.

Bei der Zeichnung der Knoten und Verbindungen soll eine schlichte Ästhetik das Ziel sein. Dazu können z.B. weiche Schatten unter den Knoten eine leichte Räumlichkeit vermitteln. Die Verbindungen

können an Stelle von einfachen Linien mit Hilfe von Splines gezeichnet werden. Die Verbindungen sollen gerade aus den Anschlüssen herausführen und sie mit einer geschwungenen Linie verbinden.

Sowohl für Schriftarten als auch für die Verbindungen sollte eine Kantenglättung aktiviert werden.

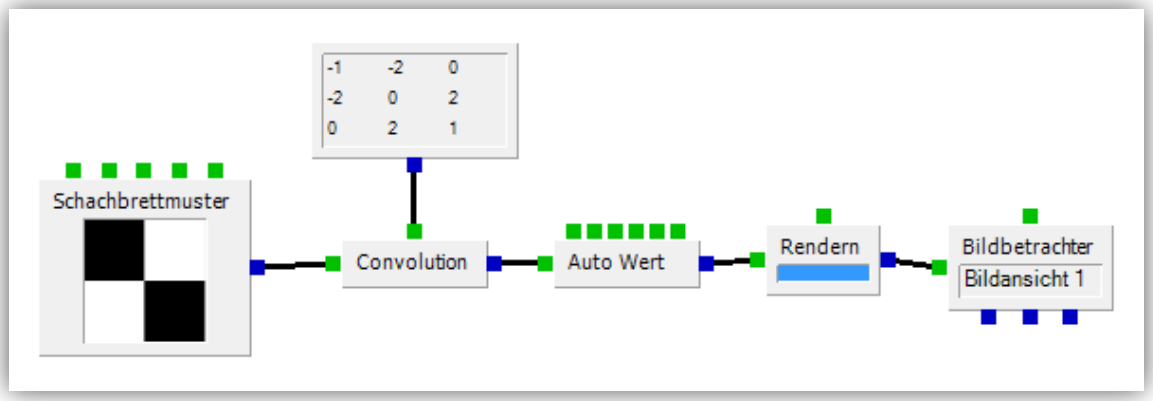


Abb. 62 Einfache grafische Darstellung eines Beispielprogramms

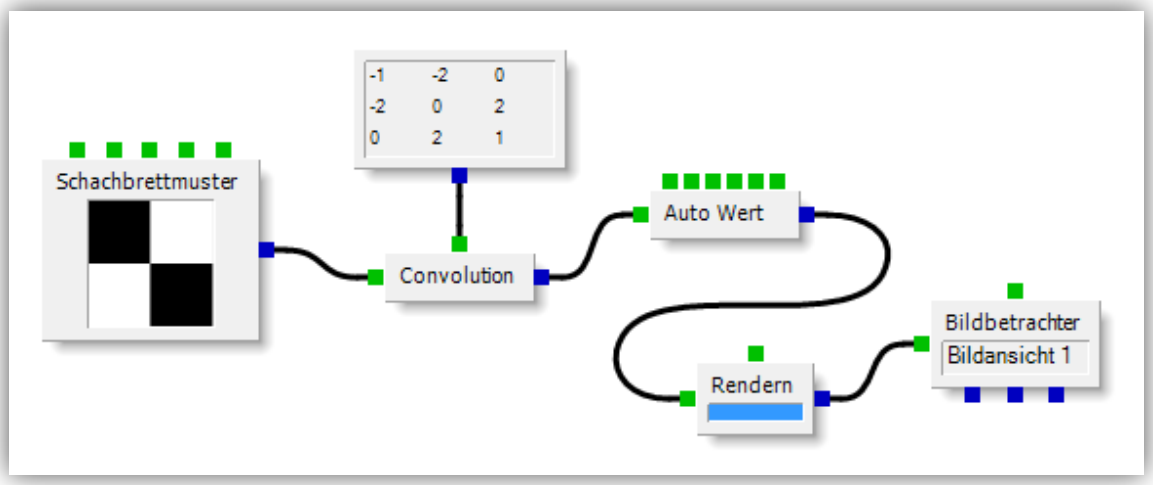
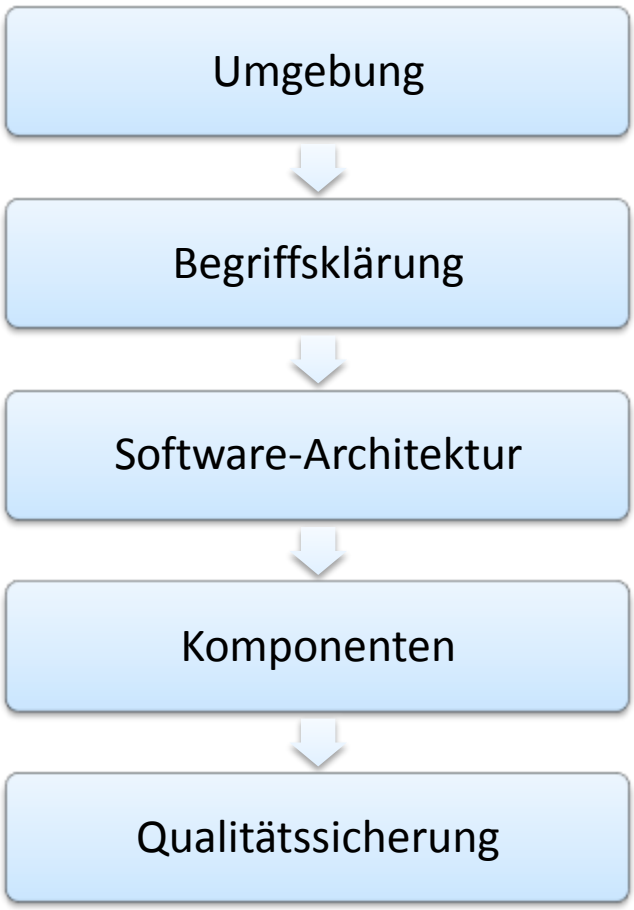


Abb. 63 Optimierte grafische Darstellung eines Beispielprogramms

5 Implementierung

Dieses Kapitel stellt eine Übersicht über die Implementierung von DynamicNodes dar. Ausgehend von der abstrakten Architektur des Systems werden die Komponenten vorgestellt aus denen DynamicNodes besteht. Anschließend werden die verschiedenen Komponenten genauer beschrieben und die Funktionen der wichtigsten Klassen erläutert. Am Ende geht ein kurzer Abschnitt auf die Qualitätssicherungsmaßnahmen ein.

Eine wichtige Referenz für dieses Kapitel ist der Anhang D mit den enthaltenen Klassenübersichten und Klassendiagrammen. Der Quelltext von DynamicNodes findet sich in ausgedruckter Form im Anhang G (Band 2) und in digitaler Form auf der beiliegenden CD.



5.1 Umgebung

Die Implementierung von DynamicNodes wurde mit Hilfe von „Microsoft Visual Studio 2005 Professional Edition“ unter „Microsoft Windows XP Professional“ durchgeführt. Als Programmiersprache kam „Visual C# 2005“ zum Einsatz. Das „Microsoft .NET Framework 2.0“ diente als Laufzeitumgebung und Klassenbibliothek.

5.2 Begriffsklärung

An dieser Stelle soll kurz die Verwendung einiger Begriffe geklärt werden.

Für die Aufteilung von Programmen in leicht zu verwaltende Blöcke, werden in der Literatur verschiedene Begriffe gebraucht. Es gibt *Komponenten*, *Module*, *Pakete*, *Namensräume* und *Assemblies*.

Ein ***Modul*** wird im Kontext der Quellcode Strukturierung als Sammlung von Prozeduren in einer Kompiliereinheit (Quellcode-Datei) verstanden. Explizit wird dieser Begriff z.B. in Visual Basic 6 und in D in diesem Zusammenhang benutzt. Da der Begriff in der .NET-Welt gewöhnlich nicht in diesem Kontext verwendet wird, soll das hier auch nicht geschehen.

Microsoft hat mit .NET den Begriff ***Assembly*** eingeführt. Er beschreibt nach (Microsoft, 2007) eine Einheit mit ausführbarem IL-Code und bildet eine Sicherheitsgrenze, eine Typgrenze, eine Versionsgrenze und eine Weitergabe-Einheit. Es gibt statische *Assemblies*, welche als PE[11] auf der Festplatte gespeichert sind und von dort vor Programmausführung in den Arbeitsspeicher geladen werden. Und es gibt dynamische *Assemblies*, welche zur Laufzeit erzeugt werden und nur im Arbeitsspeicher existieren. Dynamische *Assemblies* können als statische *Assemblies* auf die Festplatte gespeichert werden. Assemblies sind selbstbeschreibend, d.h. sie enthalten Informationen über Typen, Methoden, Felder, Bezeichner und andere Elemente des zugrunde liegenden Quellcodes.

Eine ***Komponente*** ist in der Welt des Software Engineering eine Einheit, welche definierte Schnittstellen und explizite Abhängigkeiten besitzt. Hauptmerkmal ist, dass sie austauschbar und damit komponierbar ist, d.h. sie kann mit anderen *Komponenten* zusammengefügt und ausgeführt, oder zu einer größeren *Komponente* zusammengefasst werden. Eine *Komponente* im Sinne des Software Engineering muss zu einem Komponentenmodell konform sein. In .NET ist eine *Komponente* eine Klasse, welche die Schnittstelle `System.ComponentModel.IComponent` implementiert. Alle Klassen und Schnittstellen für das entsprechende Komponentenmodell befinden sich im Namensraum `System.ComponentModel`. Da das von Microsoft vorgegebene Komponentenmodell jedoch lediglich implizit in der Implementierung der Benutzeroberfläche eingesetzt wird, wird der Begriff *Komponente* hier im Kontext der Software Architektur als Synonym für *Assembly* verwendet. Denn ein *Assembly* besitzt im Sinne der Software Architektur auch alle Merkmale einer Komponente.

[11] Portable Executable – z.B. EXE- oder DLL-Datei

Der Begriff ***Namensraum*** ist in 2.3.5 schon hinreichend beschrieben. Es sei nur erwähnt, dass *Namensräume* in der .NET-Welt eine Untergliederung von Quellcode erlauben, wobei *Namensräume* auch *Assembly*-übergreifend verwendet werden können.

Der Begriff ***Paket*** wird dagegen nur im Kontext von „Click-Once-Deployment“, also im Sinne eines Setup-Containers für lokalisierte Komponenten, verwendet. In der Programmiersprache Java hingegen sind *Pakete* und *Namensräume* ähnlich wie in der UML[12] identisch. In dieser Form werden *Pakete* auch hier betrachtet. Die in der Literatur beschriebenen und in 2.3.5 erwähnten Techniken zur Analyse und Verwaltung von *Paket*-Abhängigkeiten sollen im Rahmen dieser Arbeit jedoch auf den Kontext von *Assemblies* bzw. *Komponenten* angewendet werden. *Pakete* bzw. *Namensräume* werden innerhalb von *Assemblies* für eine thematische Gruppierung von Typen genutzt.

Nun folgen noch einige Begriffe, welche im direkten Zusammenhang mit der Programmierung mit .NET stehen.

Der Begriff ***Type*** oder *Typen* (plural) ist in .NET die Obermenge von Klassen (`class`), Schnittstellen (`interface`), Strukturen (`struct`), Aufzählungen (`enum`) und Delegaten (`delegate`).

Mit ***Eigenschaft*** (`property`) wird in .NET ein Getter-Setter-Paar bezeichnet, welches Lese- und/oder Schreibzugriff für einen benannten Wert ermöglicht. Beim Aufruf werden sie wie Felder behandelt (einer Eigenschaft kann ein Wert zugewiesen werden), bei der Deklaration ähneln sie jedoch Methoden. In C# und Visual Basic gibt es eine besondere Schreibweise für Eigenschaften, Java z.B. kennt ein besonderes Sprachkonstrukt für *Eigenschaften* hingegen nicht. In Java werden Getter und Setter als normale Methoden mit dem Präfix „Get“ oder „Is“ bzw. „Set“ deklariert.

C#

```
Datentyp Name
{
     get { return feld; }
     set { feld = value; }
}
```

Visual Basic

```
Property Name As Datentyp
     Get
          Name = feld
     End Get
     Set (ByVal value As Datentyp)
          feld = value
     End Set
End Property
```

[12] Unified Modeling Language – Ein Standard für Diagramme in der Softwaretechnik

Member (deutsch: Mitglieder) ist der Oberbegriff für nicht statische Elemente einer Klasse. Das können Felder, Methoden, Eigenschaften und Ereignisse sein.

Ein ***Delegat*** (`delegate`) ist ein typisierter Funktionszeiger. Das bedeutet, dass bei der Deklaration eines Feldes für die Speicherung eines Funktionszeiger bereits genau definiert ist, welchen Rückgabetyp und welche Parameterliste alle Funktionen haben müssen, von denen ein Zeiger in dem Feld gespeichert werden soll.

Dazu wird zunächst ein *Delegaten*-Typ deklariert:

```
delegate void EinfacheMethode(int a, int b);
```

Und anschließend kann der *Delegaten*-Typ für die Deklaration eines Funktionszeigers verwendet werden.

```
EinfacheMethode meinFunktionszeiger;
...
if (meinFunktionszeiger != null)
{
    meinFunktionszeiger(6, 7);
}
```

Ein ***Ereignis*** (`event`) in der .NET-Programmierung könnte man als *Multi-Cast-Delegat* bezeichnen. Ein *Ereignis* wird wie ein Feld jedoch mit dem Schlüsselwort `event` und einem *Delegaten*-Typ deklariert.

```
event EinfacheMethode meinEvent;
```

Einem Ereignis können mehrere Delegaten zugewiesen oder auch wieder entzogen werden.

```
meinEvent += EinErsterMethodenName;
meinEvent += EinZweiterMethodeName;
...
meinEvent(1,2); // Beide Methoden werden aufgerufen
...
meinEvent -= EinErsterMethodenName;
...
meinEvent(2, 3); // Nur die zweite Methode wird aufgerufen
```

Die GUI-Bibliothek Windows-Forms macht starken Gebrauch von *Ereignissen* um z.B. über Benutzerinteraktionen zu unterrichten. Dabei deklariert ein Steuerelement z.B. ein Ereignis mit dem Delegaten-Typ `MouseEventHandler`:

```
public event MouseEventHandler MouseClick;
```

Und der Programmierer registriert in seinem Quellcode eine Methode bei dem Ereignis:

```
steuerelement.MouseClick += MethodeDesProgrammieres;
```

Dadurch wird die Methode jedesmal aufgerufen, wenn der Benutzer auf das Steuerelement klickt.

5.3 Software-Architektur

Dieser Abschnitt soll den groben Aufbau des DynamicNode-Systems skizzieren. Es werden die Komponenten mit ihren Abhängigkeiten vorgestellt und eine Übersicht über mögliche Konfigurationen gegeben.

5.3.1 Komponenten, Abhängigkeiten

Das DynamicNode-System gliedert sich in fünf Komponenten und eine beliebige Anzahl von Knotenbibliotheken. Die Basis des Systems ist die Komponente *DNCore*. Diese enthält die Schnittstellen und Basisdatentypen für das Datenmodell der Graphen, die Laufzeitumgebung und die Persistenz. Die Komponente *DNVisual* ist die Erweiterung des Kerns, um Fähigkeiten für die grafische Ausgabe der Graphen. Sie enthält die Erweiterungen des Datenmodells für visuelle Eigenschaften und Basisklassen. Und sie enthält die Schnittstellen und Klassen, sowohl für einen grafischen Editor als auch für ein Graphen-Dokument. *DNTesting* ist eine Sammlung von Klassen zur Überprüfung der Systemkonformität von Knoten. Diese, im Vergleich zu den anderen kleine Komponente, enthält eine Testumgebung und die Schnittstellen und Basisklassen für Testfälle. Die Windows-Anwendung mit der grafischen Benutzeroberfläche ist in der Komponente *DNWinApp* untergebracht. Sie ist abhängig von allen vorher genannten Komponenten und enthält die Klassen für die Benutzeroberfläche, aber auch konkrete Implementierungen von Basisklassen und Schnittstellen aus *DNCore* und *DNVisual*. Die Komponente *DNConsole* ist ein Befehlszeilen-Tool um Graphen ohne grafische Benutzeroberfläche auszuführen.

Durch das Blockbild in Abb. 64 wird der Aufbau deutlich. Dabei ist eine Komponente von allen Komponenten abhängig, die unter ihr liegen.

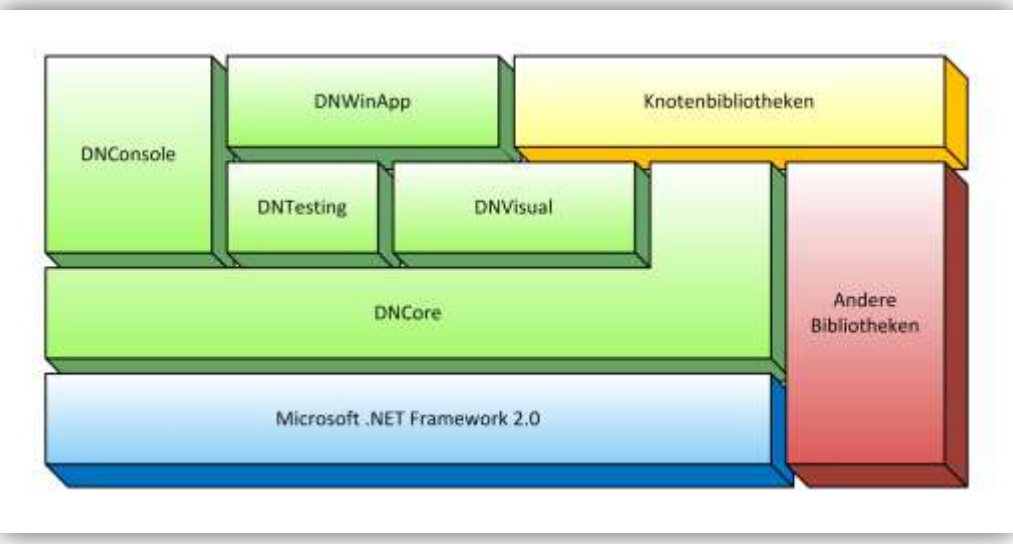


Abb. 64 Aufbau des DynamicNode-Systems

5.3.2 Schichten

An dieser Stelle soll auf die Komponentenübersichtsdiagramme im Anhang D aufmerksam gemacht werden. Sie zeigen alle Typen der verschiedenen Komponenten nach ihren Aufgaben gruppiert. Sie sind an dieser Stelle deshalb interessant, weil sich die Gruppen dieser Übersichten zum Teil in der Schichtenaufteilung wiederfinden.

DynamicNodes lässt sich wie viele Softwaresysteme in mindestens vier Schichten zerlegen. Das sind die Persistenz, die Anwendungslogik, die Präsentationlogik und die Präsentation. Im folgenden Diagramm werden Typen und Gruppen von Typen der Komponenten diesen vier Schichten zugeordnet.

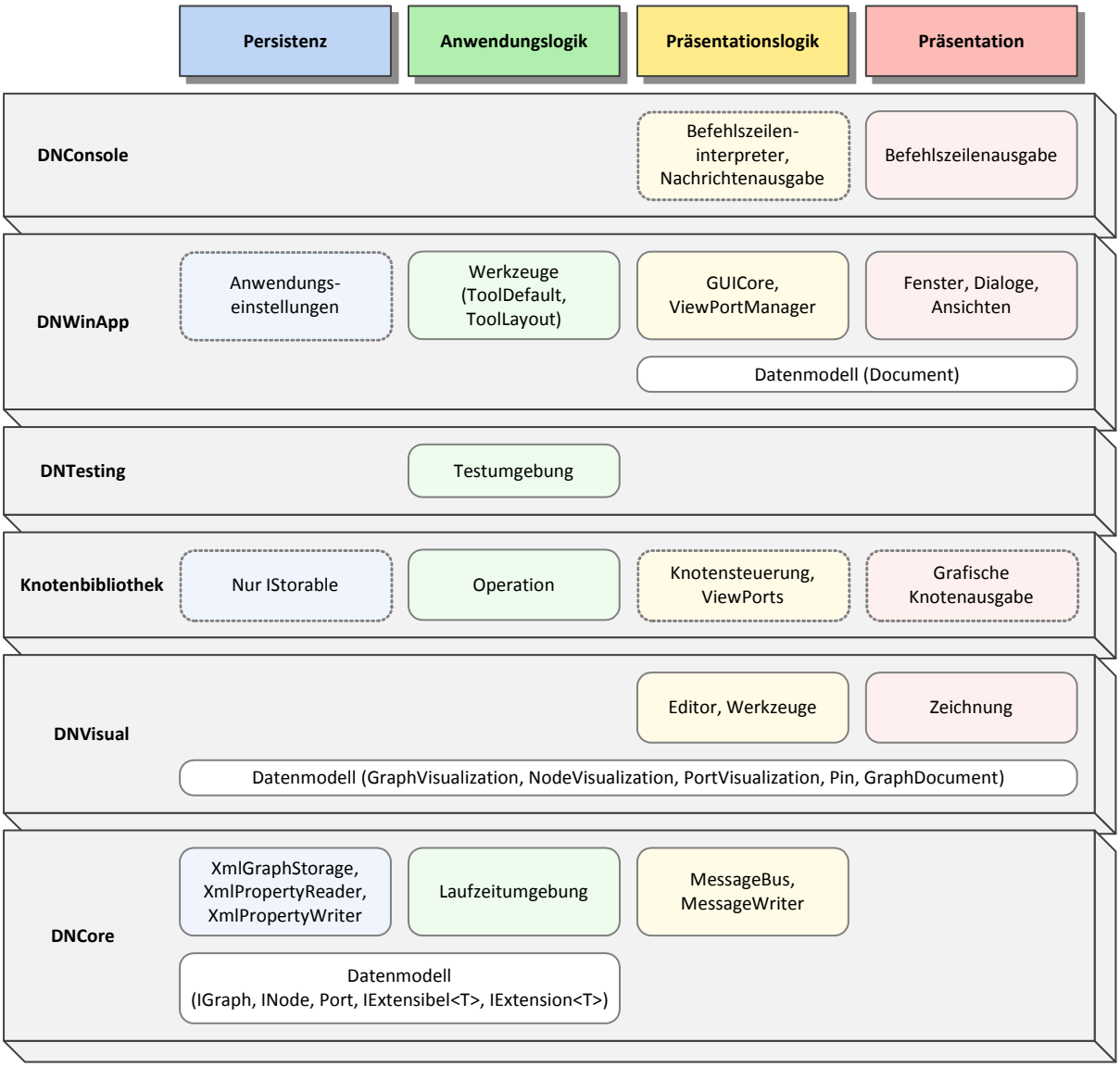


Abb. 65 Schichten in der Architektur von DynamicNodes

5.3.3 Konfigurationsmöglichkeiten

Es gibt verschiedene Möglichkeiten die Komponenten zusammenarbeiten zu lassen. Die Standard-Variante als ***Programmierumgebung*** ist wohl das Starten von *DNWinApp*. Dabei werden sowohl die Komponenten *DNCore*, *DNVisual* und *DNTesting* als auch die installierten Knotenbibliotheken geladen. Der ***Benutzer*** hat dadurch die Möglichkeit mit einem grafischen Editor Programme mit Knoten aus allen Knotenbibliotheken zu erstellen und auszuführen. Auch der Aufruf der Testumgebung aus dem Editor ist möglich.

Eine zweite Variante ist das ***Testen einer Knotenbibliothek***. Dazu wird *DNWinApp* beim Start mitgeteilt, dass es nicht den Editor sondern nur die Testumgebung anzeigen und auch nur eine bestimmte Knotenbibliothek durch die Laufzeitumgebung laden soll. Ein ***Knotenentwickler*** hat hierbei direkt die Möglichkeit entwickelte Knoten zu testen, ohne den Editor zu bemühen oder unnötig viele Knotenbibliotheken zu laden.

Wenn das Befehlszeilen-Tool *DNConsole* verwendet wird, gibt es keine grafische Benutzeroberfläche. Es werden lediglich die Komponente *DNCore* und alle installierten Knotenbibliotheken geladen. Falls eine Knotenbibliothek *DNVisual* verwendet, wird DNVisual zwar geladen, jedoch bei der Ausführung nicht benutzt. In diesem Szenario kann ein vorher erstelltes Programm auf der ***Befehlszeile***, im ***Batchbetrieb*** oder evtl. sogar als ***Dienst*** ausgeführt werden.

Es ist auch möglich *DNWinApp* derart zu starten, dass lediglich das enthaltene ***Hilfesystem*** angezeigt wird und gar keine Knotenbibliothek geladen wird.

5.4 Komponenten

In diesem Abschnitt wird auf den Aufbau der einzelnen Komponenten etwas näher eingegangen. Dabei werden die Aufgaben bzw. Rollen der wichtigsten Schnittstellen und Klassen kurz erläutert.

5.4.1 DNCore

DNCore ist die Basis für jede mögliche Konfiguration des DynamicNode-Systems. Die Klassen in *DNCore* lassen sich in folgende Gruppen unterteilen: Datenmodell, Laufzeitumgebung, Persistenz, Meta-Daten, Aggregation und Komfort. Alle Typen in *DNCore* befinden sich im Namensraum `DynamicNode.Core`. Eine Übersicht aller Typen in *DNCore* befindet sich im Anhang D - Diagramm 3 und Diagramm 4. Diagramme für die wichtigsten Typen mit ausführlicher Auflistung der Member befinden sich ebenfalls dort.

Datenmodell

In diesem Abschnitt werden alle Typen erläutert, welche unmittelbar mit dem Datenmodell von DynamicNodes zu tun haben.

Basisdatenmodell

Wie in 4.4.2 beschrieben, wird das Datenmodell hauptsächlich aus einer Klasse für die Graphen (`Graph`), einer Klasse für die Knoten (`Node`) und drei Klassen für die Anschlüsse (`Port`, `InPort`, `OutPort`) gebildet. Die Klasse `Node` ist lediglich eine abstrakte Basisklasse für alle Knotentypen, deren Implementierungen in den Knotenbibliotheken enthalten sind. `Port` ist die abstrakte Basisklasse für `InPort` und `OutPort`, und fasst deren Gemeinsamkeiten zusammen.

Für `Graph` und `Node` werden zusätzlich, und abweichend vom Entwurf, zwei Schnittstellen `IGraph` und `INode` verwendet. Diese beiden Schnittstellen beschreiben genau die öffentlichen Eigenschaften, Ereignisse und Methoden, welche `Graph` und `Node` besitzen müssen. Sie erscheinen zunächst überflüssig, ermöglichen es aber alternative Klassen für einen Graphen oder einen Knoten komplett unabhängig von der Standardimplementierung in *DNCore* zu entwickeln. Sie sind demnach lediglich eine vorausschauende Maßnahme, um die Änderbarkeit des Quellcodes zu erhöhen.

In 4.4.1 wird die Datenstruktur der Marke beschrieben. Sie findet sich im Quelltext unter dem Namen `Token` wieder. Der Aufzählungstyp für den Markenstatus trägt den Namen `TokenState`. Für die Verarbeitung von Marken-Aktualisierungen an einem Anschluss gibt es noch das Delegate `TokenUpdateHandler`, welches den Datentyp für das Ereignis `Port.TokenUpdate` darstellt.

Die Klasse `DataTypeTools` enthält hauptsächlich eine statische Methode zur Überprüfung der Verbindungs-Kompatibilität von zwei Anschlüssen. Zur Feststellung der Kompatibilität werden die priorisierten Kompatibilitätslisten in den Anschlüssen (`Port.SupportedTypes`) abgearbeitet und der Datentyp bestimmt, welcher für eine Verbindung zwischen zwei Anschlüssen am besten geeignet wäre. Dabei wird sowohl berücksichtigt, dass ausschließlich Eingänge mit Ausgängen oder umgekehrt kompatibel sind, als auch in einem Eingang nur eine Verbindung enden darf.

Das Delegate `CompatibilityChecker` ist die Definition für eine Methode, welche die Standardkompatibilität eines Ausgangs überschreiben kann. Dieses Delegate geht auf folgendes Problem zurück: Angenommen es sollte ein Eingang definiert werden, welcher Verbindungen mit beliebigen Ausgängen akzeptiert, dann könnte er einfach mit dem Datentyp `object` definiert werden. Denn alle Datentypen, die von einem Ausgang angeboten werden können, sind mit `object` in der Art kompatibel, dass eine automatische Typumwandlung nach `object` möglich ist (bei Verweistypen durch die Vererbungshierarchie und bei Werttypen durch Auto-InBoxing).

Dieser Trick ist für Ausgänge nicht möglich. Denn egal mit welchem Datentyp man einen Ausgang definiert, es wird Eingänge geben (insbesondere jene, welche mit `object` definiert sind) die dann mit diesem Ausgang nicht kompatibel sind. Eine offensichtliche Lösung scheint das Definieren aller Ein- und Ausgänge mit dem Datentyp `object`. Jedoch hebelt dieses Vorgehen die Typensicherheit von DynamicNodes komplett aus.

Die Lösung für das Dilemma ist die Möglichkeit, dass der Knotenentwickler die Kompatibilitätsprüfung von Ausgängen aktiv beeinflussen kann. Und genau das ist mit dem Feld `Out-`

`Port.CompatibilityChecker` vom Typ `CompatibilityChecker` möglich. Denn sobald ein Knotenentwickler diesem Feld bei der Erzeugung eines Ausgangs eine eigene Methode zuweist, wird die Kompatibilitätsprüfung des Ausgangs nicht mehr automatisch mit der Kompatibilitätsliste durchgeführt, sondern mit Hilfe der vom Knotenentwickler implementierten Methode.

Erweiterung

Die im Entwurf beschriebene Erweiterung des Basisdatenmodells durch generische Schnittstellen wurde mit den beiden Klassen `IExtensible<T>` und `IExtension<T>` durchgeführt. Dabei beschreibt `IExtensible<T>` eine Beziehung zu Objekten, welche `IExtension<T>` implementieren. In der praktischen Anwendung dieser Schnittstellen wird `IExtensible<T>` von einer Klasse des Datenmodells implementiert und eine Erweiterungsklasse implementiert `IExtension<T>`. `INode` z.B. ist von `IExtensible<INode>` abgeleitet und kann deshalb eine Beziehung zu Objekten aufbauen, welche `IExtension<INode>` implementieren.

Ereignisse

Die Schnittstelle und Klassen des Datenmodells deklarieren eine Reihe von Ereignissen, welche über Änderungen des Datenmodells informieren. So besitzt Port z.B. das Ereignis `ConnectionChanged`, welches ausgelöst wird, wenn die Anzahl der Verbindungen zu diesem Anschluss geändert wurde.

Die meisten Ereignisse entsprechen dabei der Microsoft-Empfehlung und verwenden das generische Delegate `EventHandler<T>`, welches die Methoden-Signatur `void Methode(object sender, T eventArgs)` definiert, als Datentyp. Die Ereignisse der Windows-Forms-Bibliothek sind ausnahmslos nach diesem Schema aufgebaut. Dabei ist `sender` ein Verweis auf das Objekt, welches das Ereignis ausgelöst hat und `eventArgs` ist ein Verweis auf eine Kind-Klasse von `System.EventArgs`, welche die Ereignis-spezifischen Parameter kapselt.

Für das Datenmodell von DynamicNodes gibt es die Klassen `GraphEventArgs`, `NodeEventArgs` und `ConnectionEventArgs`, welche entsprechende Parameter kapseln.

Laufzeitumgebung

Die Typen für die Laufzeitumgebung lassen sich in drei Gruppen gliedern. Die allgemeinen Typen für die Infrastruktur (Basis), die Typen für den Nachrichten-Bus und die Typen für die Verwaltung der Parallelität.

Basis

Der „Kern vom Kern“ ist die Schnittstelle `IRuntime`. Sie definiert alle Eigenschaften und Methoden für den Knotenpunkt der Laufzeitumgebung. Ihre Standardimplementierung ist die Klasse `Runtime`. Diese Klasse erfüllt zwei Aufgaben: Das Bereitstellen von Verweisen auf die unterschiedlichen Klassen, aus denen sich die Laufzeitumgebung zusammensetzt, und das Verwalten der Graphen, welche ausgeführt werden sollen. Nahezu alle Objekte im Objektgraph von DynamicNodes besitzen direkt oder indirekt einen Verweis auf eine `Runtime`-Instanz. Gewöhnlich gibt es in einer Anwendung nur

eine `Runtime`-Instanz. Dennoch ist die Klasse bewusst nicht als Singleton implementiert, damit die Möglichkeit besteht in einer *Application-Domain* mehrere Laufzeitumgebungen zu benutzen.

Die Schnittstelle `IConfiguration` und ihre Standardimplementierung `Configuration` kapseln eine Reihe von Dateisystem-Pfaden. Dazu gehört u.a. das Verzeichnis mit den Knotenbibliotheken.

Die Klasse `NodeFactory` ist für das Laden von Knotenbibliotheken und das Instanziieren von Knoten zuständig. Sie durchsucht während der Initialisierungsphase die Verzeichnisse für die Knotenbibliotheken nach DLL-Dateien und versucht diese als .NET-Assembly zu laden. Anschließend werden alle erfolgreich geladenen Assemblies nach Klassen durchsucht, welche die Schnittstelle INode implementieren, nicht abstrakt sind und einen parameterlosen Konstruktor besitzen. Diese Klassen werden für die Laufzeit als Knotentyp vorgemerkt.

Parallelität

Für die Verwaltung von Threads zur Ausführung von parallelen Knoten ist die Schnittstelle `IThreadManager` zuständig. Während der Entwicklung von DynamicNodes sind zwei Implementierungen entstanden. `ThreadManager` versucht die Threads manuell zu verwalten. Die Klasse `PoolThreadManager` benutzt die .NET-Klasse `System.Threading.ThreadPool` für die Verwaltung eines Thread-Pools. Sie ist zur Zeit die bessere Wahl und wird von DynamicNodes als Standard-Thread-Manager verwendet.

Nachrichten-Bus

`IRuntime` besitzt das Feld `IRuntime.MsgBus`, welches einen Verweis auf ein Objekt vom Typ `MessageBus` hält. Dieses Objekt ist über `IRuntime` von nahezu allen Objekten im Objektgraph erreichbar und wird verwendet, um Nachrichten an den Benutzer zu schicken. Diese Nachrichten werden mit Hilfe der Struktur `Message` gebildet und durch `MessageTyp` in vier Gruppen unterschieden: `Debug`, `Info`, `Warning`, `Error`.

`MessageBus` ist die zentrale Anlaufstelle für alle Nachrichten und löst bei Eintreffen einer Nachricht das Ereignis `MessageBus.Message` aus. Die Klasse `MessageWriter` kann dieses Ereignis mit `MessageWriter.WriteMessage(Message msg)` verarbeiten und in Textform z.B. auf der Konsole oder in einer Log-Datei ausgeben.

Persistenz

Graphen mit ihren Knoten und Verbindungen werden mit Hilfe von Objekten gespeichert und wieder hergestellt, welche die Schnittstelle `IGraphStorage` implementieren. Für die Speicherung und Restauration von Eigenschaftslisten werden dabei Objekt verwendet, welche die Schnittstellen `IPropertyWriter` und `IPropertyReader` implementieren. DynamicNodes verwendet dabei das XML-Format und die Klassen `XmlGraphStorage`, `XmlPropertyWriter` und `XmlPropertyReader`. Jedoch steht durch die Abstraktion mit Hilfe der Schnittstellen der Verwendung einer alternativen Speichertechnik (z.B. mit SQL oder OO-Datenbanken) nichts im Weg. Das konkrete XML-Format für

DynamicNodes-Graphen wird in der Datei `Project.xsd` definiert, welche im Anhang G - Projects.xsd (Band 2, S. 442) enthalten ist.

Meta-Daten

Es gibt lediglich zwei Klassen die verwendet werden, um die Meta-Daten von Knotentypen direkt zu unterstützen: `NodeInfoAttribute` und `StaticNodeInfoAttribute`. Wie die Endung `...Attribute` bereits andeutet, handelt es sich bei diesen Klassen um Attribute (in der Java-Welt als Annotationen bekannt), mit deren Hilfe Code-Elemente wie Typen, Eigenschaften oder Methoden mit Meta-Daten angereichert werden können. Hier werden sie verwendet, um beschreibende Informationen für einen Knotentyp bereitzustellen, ohne dass eine Instanziierung des Knotentyps nötig wird.

Die `KlasseNodeInfoAttribute` bietet die Möglichkeit direkt drei Zeichenketten zur Benennung, Gruppierung und Beschreibung an eine Knotenklasse anzufügen. Da Attribute als Parameter nur Konstanten entgegennehmen, ist mit dieser Variante jedoch keine Lokalisierung (mehrsprachige Bereitstellung) der Meta-Daten möglich. Mit der Klasse `StaticNodeInfoAttribute` hingegen kann eine statische Methode als Quelle für die Meta-Daten angegeben werden. Mit einer statischen Methode ist es kein Problem z.B. eine lokalisierte Ressource für die Meta-Daten zu verwenden.

Aggregation

Die Aggregation von Graphen zu Knoten wird mit der Klasse `NSubGraph` ermöglicht. Sie ist eine vollständige Implementierung der `INode`-Schnittstelle und leitet sich von `Node` ab. `NSubGraph` besitzt die Fähigkeit einen Graphen aus einer XML-Datei zu laden und deren Super-Ein- und Super-Ausgänge als eigene Ein- und Ausgänge nach außen zu führen. Dabei kommen die Klassen `SuperInPortWrapper` und `SuperOutPortWrapper` zum Einsatz. Sie sind jeweils von `InPort` und `OutPort` abgeleitet und leiten alle notwendigen Methodenaufrufe an die Super-Anschlüsse im aggregierten Graphen weiter.

Damit die Marken von den Super-Ausgängen des Sub-Graphen korrekt an den zuständigen `SuperOutPortWrapper` weitergeleitet werden, wird die Klasse `SuperOutPortExtension` als Anschluss-Erweiterung an den entsprechenden Super-Ausgang im Sub-Graph angefügt.

Komfort

Um die Entwicklung von Knotentypen zu vereinfachen, gibt es eine Gruppe von Komfort-Typen. Die Klassen `InPortInfoAttribute` und `OutPortInfoAttribute` dienen dazu eine Ein- bzw. Ausgang mit Hilfe eines Attributs zu definieren. Für gewöhnlich muss ein Knotenentwickler die Anschlüsse seines Knotens im Konstruktor der Knotenklasse instanziieren und dabei an den Knoten anfügen. Mit den oben genannten Klassen ist das nicht notwendig. Um einen Eingang zu definieren, genügt es ein öffentliches Feld oder eine öffentliche Eigenschaft mit dem `InPortInfoAttribute` zu versehen, um das Feld bzw. die Eigenschaft mit einem automatisch instanziierten Eingang zu verknüpfen. Für einen Ausgang wird dabei ein öffentliches Feld vom Typ `OutPortHandler` verwendet. Ein prak-

tisches Beispiel für diese Vorgehensweise findet sich im Anhang F - Einstieg in die Knotenentwicklung.

5.4.2 DNVisual

Die Komponente *DNVisual* enthält die Erweiterung des Basis-Datenmodells um die visuellen Daten, die Grundlage für den grafischen Editor zur Bearbeitung von Graphen, die Grundlage für Werkzeuge zur Verwendung im Editor, einen Mechanismus zur Synchronisation von Threads mit dem GUI-Thread und zwei Klassen zur komfortablen Entwicklung von Knotentypen, welche die visuellen Erweiterungen explizit verwenden. Alle Typen in DNVisual befinden sich im Namensraum `DynamicNode.Ext.Visual`. Eine Übersicht ist im Anhang D - Diagramm 24 enthalten.

An dieser Stelle ist es günstig das Dreiergespann aus `IEditor`, `IGraphDocument` und `IPainter` zu erwähnen. Diese drei Typen bilden die Grundlage für die Umsetzung des MVC-Modells[13] in der DynamicNodes-Architektur.

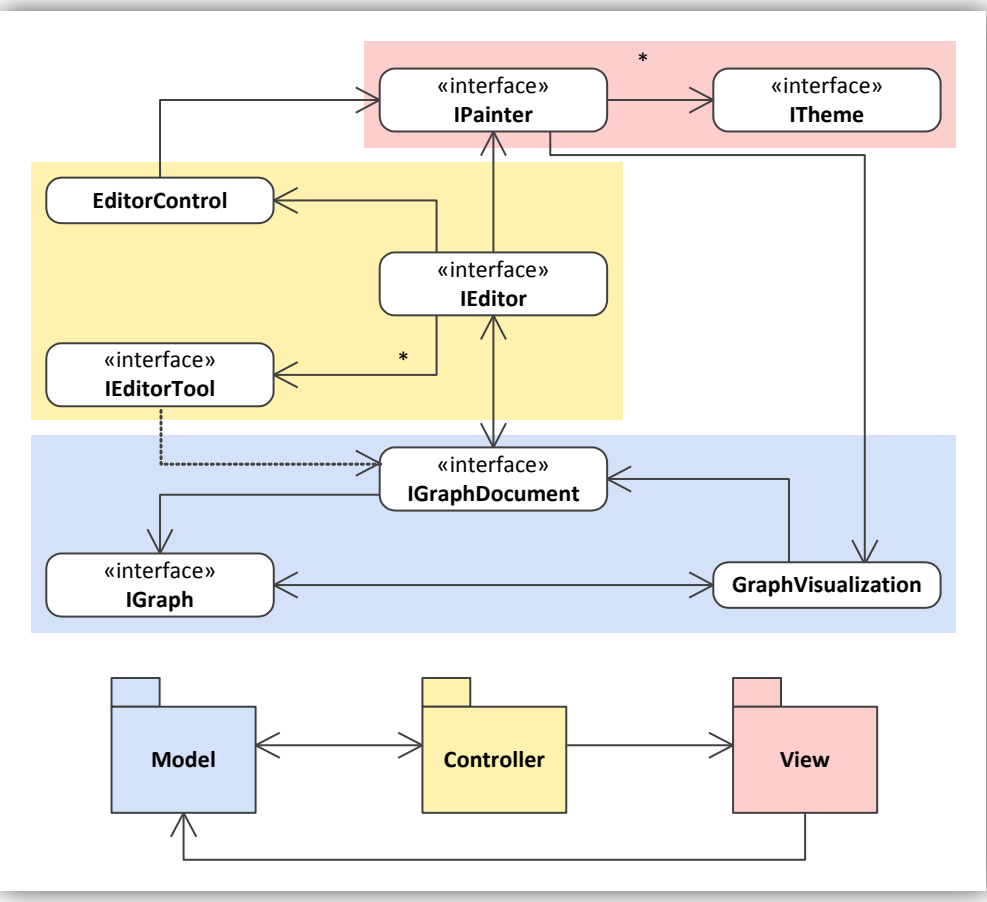


Abb. 66 Das MVC-Modell in DynamicNodes

[13] Model View Controller – Ein weit verbreitetes Entwurfsmuster für die Architektur von Software mit Benutzerschnittstelle

Die Klasse `GraphDocument` mit der Schnittstelle `IGraphDocument` bildet mit einem Verweis auf das zugehörige `IGraph`-Objekt, welches wiederum die visuelle Erweiterung `GraphVisualization` besitzt, das ***Model***.

`IEditor` bzw. die Standardimplementierung `Editor` bildet mit dem Steuerelement `EditorControl` und den Implementierungen von `IEditorTool` den ***Controller***. Der Editor verarbeitet dabei Benutzerereignisse von der View vor und gibt sie an `IGraphDocument` weiter. Dieser löst daraufhin entsprechende Ereignisse aus, welche von den Implementierungen von `IEditorControl` empfangen und in Manipulationen an `IGraphDocument` umgesetzt werden.

`IPainter` bzw. die Standardimplementierung `Painter` bildet mit den Implementierungen von `ITheme` die ***View*** und sorgt für die grafische Darstellung eines Graphen unter Berücksichtigung des aktuellen Bearbeitungszustandes (Zoom, laufende Drag&Drop-Operationen). Der Zeichenvorgang kann sowohl explizit vom *Controller* als auch implizit durch Ereignisse am *Model* gestartet werden.

Datenmodell

Die Erweiterung des Basis-Datenmodells wird hauptsächlich durch die Klassen `GraphVisualization`, `NodeVisualization` und `PortVisualization` gebildet. Diese drei Klassen sind von `IExtensible<T>` abgeleitet. Die Klasse `Pin` wird von `PortVisualization` benutzt, um Zwischenpunkte einer Verbindung zur grafischen Umlenkung zu speichern.

Um einen Graphen mit seinen visuellen Erweiterungen als Ganzes besser verwalten zu können, wird an dieser Stelle die Klasse `GraphDocument` eingeführt. Sie ist das Datenmodell für den Editor bzw. den Arbeitsbereich, welcher im Abschnitt 4.6.4 beschrieben wird.

`GraphDocument` übernimmt mehrere Aufgaben. Eine Aufgabe besteht darin, die Ereignisse, welche vom Editor durch Benutzerinteraktion ausgelöst werden, anderen Objekten zentral zugänglich zu machen. Eine weitere Aufgabe ist es, Zustände wie die Auswahl von Knoten oder Ports zu speichern. Die Fähigkeiten dieser Klasse sind durch die Schnittstelle `IGraphDocument` genau festgelegt.

Synchronisation

Die Benutzeroberflächenbibliothek Windows-Forms im .NET-Framework hat die Einschränkung, dass Steuerelement nur innerhalb des Threads manipuliert werden können, von dem sie erzeugt wurden. Da die Benutzerschnittstelle von DynamicNodes von einem Haupt-Thread erzeugt wird, könnten Knoten, welche parallel in anderen Threads arbeiten, keine grafischen Ausgaben erzeugen.

Die Lösung für diese Einschränkung ist ein kleiner Trick. Windows-Forms bietet die Möglichkeit dem Haupt-Thread eine Methode mit Hilfe eines Delegates für die Ausführung zu übergeben. Um diese Möglichkeit für den Knotenentwickler komfortabel zu kapseln, bringt die Komponente *DNVisual* die Klasse `ThreadHelper` mit. Durch einen Aufruf der mehrfach überladenen Methode `ThreadHelper.SaveInvoke(…)` kann die Benutzeroberfläche aus einem fremden Thread heraus manipuliert werden.

Benutzeroberfläche

Einige Typen in *DNVisual* lassen sich direkt der Benutzeroberfläche zuordnen. Dazu zählen z.B. die Klasse `ViewPortExtension`, mit deren Hilfe ein Knoten einen spezifischen Ausgabebereich anfordern kann. Oder die Klasse `ControlPanelExtension`, mit deren Hilfe ein Knotenentwickler die Benutzerschnittstelle der Knotensteuerung für einen Knotentyp definiert.

Die Fähigkeiten der oben bereits erwähnte Klasse `Editor` werden durch die Schnittstelle `IEditor` definiert. Das tatsächliche Steuerelement, welches den Arbeitsbereich des Editors auf dem Bildschirm repräsentiert, ist `EditorControl`.

Ein Anzahl von Ereignis-Argumenten gehören mit zum Editor: `EditorMouseEventArgs`, `EditorDragEventArgs`, `EditorNodeEventArgs` und `EditorPortEventArgs`.

Für Farben und Formen der grafischen Ausgabe sind die Schnittstellen `IPainter` mit der Standardimplementierung `Painter` und `ITheme` mit der Standardimplementierung `RoundedTheme` verantwortlich. Die Klasse `PaintSettings` enthält das Farbschema für die Darstellung der Graphen.

Werkzeuge

Die Editor-Werkzeuge bilden den Controller des MVC-Modells. Die Anforderungen an ein Editor-Werkzeug sind durch `IEditorTool` beschrieben. Die abstrakte Implementierung `AbstractTool` vereinfacht die konkrete Entwicklung eines Werkzeugs, welches von `AbstractTool` abgeleitet werden kann.

Komfort

Um die Entwicklung von Knotentypen zu vereinfachen, welche ihre grafische Ausgabe beeinflussen oder die Benutzerschnittstelle für eine Knotensteuerung mitbringen, kann die abstrakte Klasse `VisualNode` als Basisklasse verwendet werden.

Ein Knotentyp, der einen Fortschrittsbalken auf der Arbeitsfläche anzeigt, kann auf der Grundlage von `ProgressNode` sehr einfache implementiert werden.

5.4.3 DNTesting

Die Komponente DNTesting beinhaltet die Testumgebung und einige Beispiel-Testfälle. Eine Übersicht über alle Typen der Komponente DNTesting findet sich in Anhang D - Diagramm 38.

Testumgebung

Die Testumgebung befindet sich im Namensraum `DynamicNode.Test`. Die Klasse `TestBench` bildet im Prinzip die gesamte Testumgebung. Sie führt an einem Knotentyp automatisch alle registrierten Testfälle durch und informiert mittels Ereignissen über die Ergebnisse.

Die beiden Gruppen von Testfällen (Typen-Tests und Instanz-Tests) werden durch die Schnittstellen `INodeInstanceTestCase` und `INodeTypeTestCase` repräsentiert. Beide Schnittstellen erben gemeinsame Fähigkeiten von `ITestCase`. Die Aufzählung `TestEventTyp` dient dazu, die Ergebnisse der Tests nach dem Grad des Erfolges zu unterscheiden. Dabei definiert die Aufzählung drei Gruppen: `Info`, `Warning` und `Error`.

Testfälle

Die Beispieltestfälle befinden sich im Namensraum `DynamicNode.Test.Cases` und sind von den abstrakten Basis-Klassen `BaseInstanceTestCase` bzw. `BaseTypeTestCase` abgeleitet, welche wiederum die oben genannten Schnittstellen `INodeInstanceTestCase` und `INodeTypeTestCase` implementieren.

Die Beispiel-Testfälle sind in Abb. 67 aufgelistet.

Klasse	Test-Typ	Aufgabe
TCreation	Typen-Test	Überprüft die Instanziierung des Knotentyps
TCreationVisual	Typen-Test	Überprüft die Unterstützung für visuelle Erweiterungen
TNames	Typen-Test	Überprüft, ob sinnvolle Meta-Daten verwendet werden
THelp	Typen-Test	Überprüft das Vorhandensein und die Vollständigkeit der Knotenbeschreibung für die Knotenreferenz
TPaint	Instanz-Test	Überprüft, ob bei der grafischen Ausgabe Fehler auftreten

Abb. 67 Tabelle mit Beispiel-Testfällen

5.4.4 DNWinApp

Die Komponente *DNWinApp* bildet in Form einer EXE-Datei die Windows-Anwendung, welche alle anderen Komponenten (außer DNConsole) als DLL referenziert. In *DNWinApp* werden fünf Namensräume zur Gruppierung der Typen verwendet. *DNWinApp* enthält hauptsächlich Typen für die Benutzeroberfläche von DynamicNodes.

Alle Typen von *DNWinApp* sind im Anhang D - Diagramm 45 und Diagramm 46 in einer Übersicht dargestellt. Eine Liste mit den Befehlszeilen-Parametern von DNWinApp findet sich ebenfalls in Anhang E - Tabelle 2.

Einen Screenshot der DynamicNodes-Windows-Anwendung unter Microsoft Windows Vista zeigt Abb. 68.

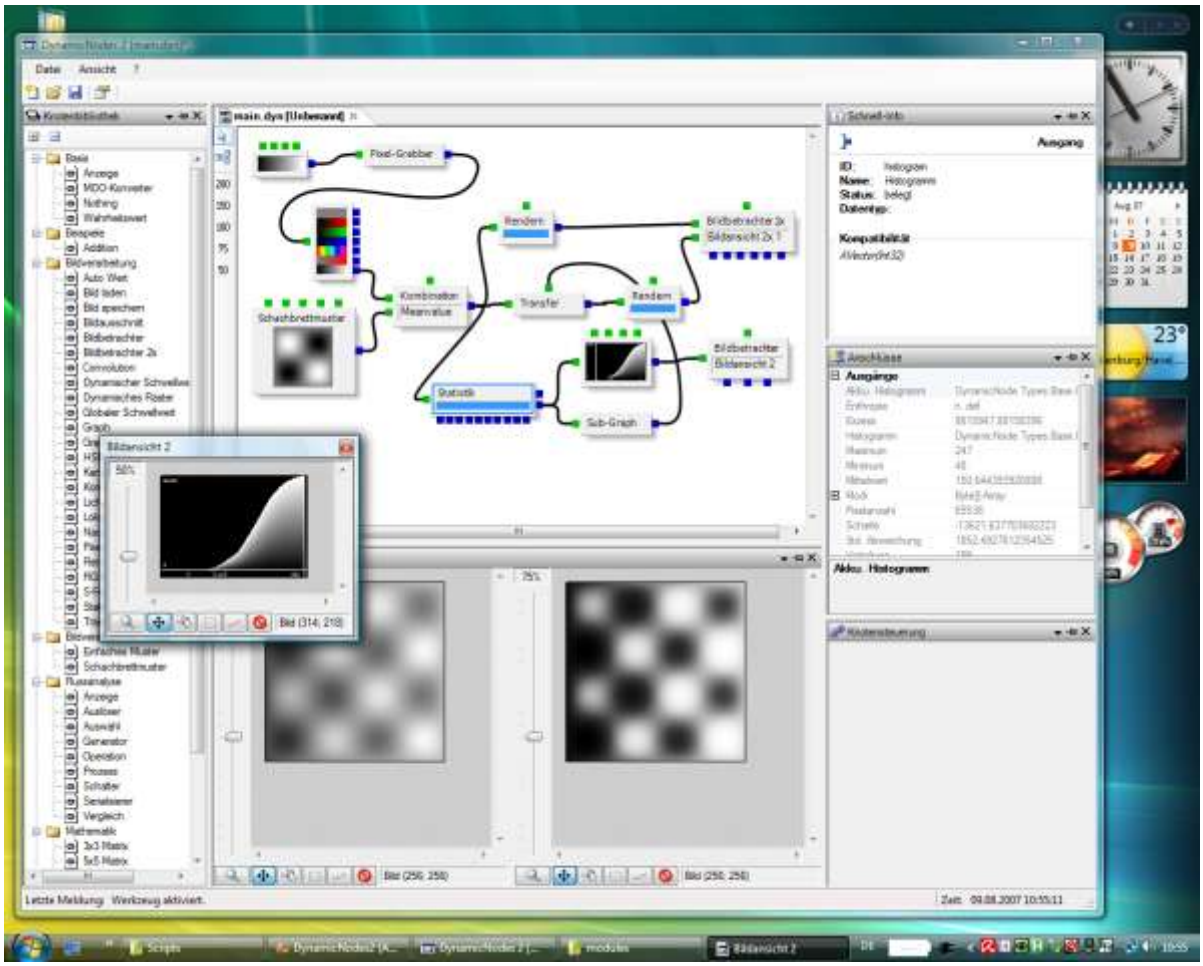


Abb. 68 DynamicNodes unter Windows Vista

Allgemein

Der allgemeine Namensraum in *DNWinApp* ist `DynamicNode.WinApp`. Er lässt sich wiederum in vier Bereich aufteilen: Typen die dem Hilfesystem zuzuordnen sind, Typen die der Testumgebung zuzuordnen sind, Typen des Datenmodells und alle übrigen für die Windows-Anwendung relevanten Typen.

Die Klasse `Program` besitzt die statische Methode `void Main(string[] args)` und ist damit der Einsprungspunkt für das Programm. Sie gehört zu der letzten oben genannten Gruppe. Dazu gehören auch die Klasse `StartupSettings`, welche die Befehlszeilenargumente des Programmaufrufs auswertet, ebenso das Hauptfenster `MainForm`, das Begrüßungsfenster `SplashForm` und das Informationsfenster `InfoDialog`. Weitere Dialoge sind der `SettingsDialog`, welcher mit dem PropertyGrid-Steuerelement die Konfiguration von DynamicNodes ermöglicht, und der `PropertyDialog`, welcher ebenfalls mit dem PropertyGrid-Steuerelement für die Eigenschaften von Graphen, Knoten und Anschlüssen zuständig ist.

Die Klasse `ViewPortManager` verwaltet die Knoten-spezifischen Ausgabebereiche als Ansichten in der Benutzeroberfläche. Dabei wird die Klasse `ViewPort` als Container-Ansicht verwendet.

In der Gruppe des Datenmodells findet sich die Klasse `Document`. Sie erbt von `DynamicNode.Ext.Visual.GraphDocument` und erweitert es um die Fähigkeit der Integration in das MDI (Multi-Document-Interface) der Benutzeroberfläche. Dadurch wird die Klasse `Document` zum zentralen Datenmodell für jedes DynamicNodes-Programm im Kontext des visuellen Editors. Die Klasse `GraphListener` überwacht die Ereignisse der verschiedenen DynamicNodes-Programme in der Laufzeitumgebung und sorgt bei Bedarf für eine Aktualisierung der Benutzeroberfläche.

Die Zentrale des Datenmodells für die Benutzeroberfläche ist die Klasse `GUICore`. Sie besitzt Verweise auf die wichtigsten Objekte der Benutzeroberfläche und stellt diese anderen Klassen als öffentliche Eigenschaften zur Verfügung.

Das Hilfesystem wird hauptsächlich durch die Klassen `HelpControl` und `HelpContentReader` implementiert. `HelpContentReader` durchsucht das Hilfe-Verzeichnis nach gültigen HTML-Hilfedateien und XML-Knotenbeschreibungen und erzeugt damit einen Navigationsbaum. Das Steuerelement `HelpControl` entspricht der Benutzerschnittstelle in 4.6.3 - Grafische Aufteilung - Hilfesystem und zeigt links den Navigationsbaum und rechts die Hilfe-Inhalte an. Dabei kommt ein Web-Browser-Steuerelement zum anzeigen der Inhalte zum Einsatz. Die XML-Knotenbeschreibungen werden dabei automatisch mittels XSLT[14] in HTML transformiert. Das Format der XML-Knotenbeschreibungen ist durch das XML-Schema `Description.xsd` (Anhang G - Description.xsd; Band 2, S. 444) vorgegeben, die Transformation wird mittels `Description.xslt` (Anhang G - Description.xslt; Band 2, S. 446) durchgeführt.

Für das Datenmodell des Navigationsbaums kommen noch einige Hilfsklassen zum Einsatz: `LibraryDescriptor`, `GroupDescriptor` und `NodeDescriptor`.

Wenn DNWinApp mit dem Befehlszeilen-Parameter `-help` aufgerufen wird, sorgt die Main-Methode in `Program` dafür, dass statt des Hauptfensters `MainForm` das Fenster `HelpForm` angezeigt wird. Dieses enthält nichts weiter als das Steuerelement `HelpControl` und stellt damit eine Möglichkeit dar, das Hilfesystem unabhängig vom Editor zu starten.

Im Gegensatz zum Hilfesystem, welches mit Hilfe von `HelpControl` und `ViewHelp` auch im Hauptfenster angezeigt werden kann, wird die Testumgebung immer mit einem eigenen Fenster gestartet. Das Fenster der Testumgebung ist `TestForm`. Es entspricht dem Layout aus 4.6.3 - Grafische Aufteilung - Testumgebung und ruft bei Bedarf `TestEventItemDialog` auf, um die Details eines Test-Ergebnisses anzuzeigen.

PropertySupport

Der Namensraum `DynamicNode.WinApp.PropertySupport` enthält Klassen zur Erweiterung des PropertyGrid-Steuerelements (`System.Windows.Forms.PropertyGrid`). Das PropertyGrid-

[14] XSL Transformations – Extensible Stylesheet Language Transformation – Eine XML-Sprache zur automatischen Transformation von XML-Dokumenten

Steuerelement basiert auf dem .NET-Component-Model und zeigt eine grafische Benutzerschnittstelle für die Anzeige und Manipulation von öffentlichen Objekt-Eigenschaften.

DynamicNodes bringt die Unterstützung für zwei Typen mit: `System.IO.FileInfo` und `System.IO.DirectoryInfo`. Um .NET-Component-Model-konforme Unterstützung für diese zwei Typen zu bieten, müssen Klassen erweitert werden. Zu diesen Klassen gehören `System.ComponentModel.TypeDescriptionProvider`, `System.ComponentModel.CustomTypeDescriptor`, `System.Drawing.Design.UITypeEditor` und `System.ComponentModel.TypeConverter`. Der Erweiterung dienen die Klassen `FileInfoProvider` und `DirectoryInfoProvider` und ihre inneren Klassen. `PropertySupport` bietet eine statische Methode `PropertySupport.Initialize()`, welche die Typen-Unterstützung für `FileInfo` und `DirectoryInfo` beim Component-Model von .NET registriert.

PropertyWrapper

Ebenfalls mit dem PropertyGrid-Steuerelement hat der Namensraum `DynamicNode.WinApp.PropertyWrapper` zu tun. In 4.6.6 wurde festgelegt, dass ein generischer Eigenschaftsdialog mit Hilfe des PropertyGrid-Steuerelements die Eigenschaften von unterschiedlichen Objekten zugänglich machen soll. Jedoch sind nicht immer alle Eigenschaften auf Quellcode-Ebene für den Benutzer interessant oder eine Manipulation sinnvoll. Deshalb sollen sog. Wrapper-Klassen verwendet werden, um eine ausgewählte Anzahl von Eigenschaften durch das PropertyGrid-Steuerelement für den Benutzer zugänglich zu machen.

Der Nachrichten-Bus (`DynamicNode.Core.MessageBus`) besitzt z.B. wesentlich mehr Eigenschaften und Methoden als für den Benutzer interessant sind. Ihn interessiert lediglich die Einstellung, ob bei der Protokollierung der Ursprung von Nachrichten im Quellcode protokolliert werden soll (`MessageBus.TraceLocation`). Nun wird dem PropertyGrid-Steuerelement nicht das echte `MessageBus`-Objekt übergeben sondern eine Instanz von `MsgBusWrapper`. Diese Instanz kapselt das echte `MessageBus`-Objekt und führt nur die Eigenschaft `TraceLocation` nach außen. Dabei wird die öffentliche Eigenschaft `MsgBusWrapper.TraceLocation` mit lokalisierten Meta-Daten versehen, so dass das PropertyGrid-Steuerelement die Beschriftung und Beschreibung der Eigenschaft in einer für den Benutzer verständlichen Weise darstellen kann.

Für die lokalisierten Meta-Daten von Eigenschaften werden die Klassen `DNCategoryAttribute`, `DNPropertyDescriptor` und die Aufzählung `DNCategoryTyp` verwendet.

Um die Ein- und Ausgänge eines Knotens in einem PropertyGrid-Steuerelements anzeigen und manipulieren zu können, müssen sie als Eigenschaften eines Objektes präsentiert werden. Dazu dienen die Klassen `InPortPropertyDescriptor` und `OutPortPropertyDescriptor` zusammen mit der Klasse `NodePortWrapper`. Die beiden PropertyDescriptor-Klassen gaukeln dem Component-Model von .NET vor, dass die Klasse `NodePortWrapper` immer genau jene öffentlichen Eigenschaften besitzt, welche zur Darstellung der Ein- und Ausgänge eines Knotens erforderlich sind.

Neben `MsgBusWrapper` und `NodePortWrapper` enthält der Namensraum `DynamicNode.WinApp.PropertyWrapper` noch eine ganze Reihe weiterer Wrapper, welche alle von der abstrakten Hilfsklasse `AbstractWrapper` abgeleitet sind.

Werkzeuge

Die Komponente DNWinApp bringt zwei konkrete Editor-Werkzeuge mit. Diese befinden sich im Namensraum `DynamicNode.WinApp.Tools` und heißen ToolDefault und ToolLayout. Die beiden Werkzeuge unterscheiden sich dahingehend, dass mit ToolLayout keine Verbindungen zwischen Knoten erzeugt oder zerstört werden können, dafür aber grafische Umlenkungspunkte (Pins) erzeugt, zerstört und verschoben werden können.

Als Unterstützung für die Werkzeuge existiert eine Klasse `ContextMenuFactory`, welche dynamisch Kontext-Menüs für Knoten, Anschlüsse und Pins generiert und anzeigt.

Ansichten

Für die flexible Aufteilung der Benutzeroberfläche in DynamicNodes wird wie in 4.6.5 beschrieben die DockSystem-Bibliothek des Autors verwendet. Mit deren Hilfe können die verschiedenen Benutzerschnittstellen als Ansicht definiert werden. Der Namensraum `DynamicNode.WinApp.Views` enthält die verschiedenen Ansichten der DynamicNodes-Benutzeroberfläche.

Alle Ansichts-Klassen sind von der abstrakten Klasse `AbstractView` abgeleitet. Dazu zählen `ViewHelp`, `ViewLog` und `ViewNodeLibrary`. Eine Besonderheit bilden die abstrakte Klasse `AbstractNodeView` und deren Kind-Klassen. Sie werden automatisch über die Auswahl eines Knotens informiert und können ihren Inhalt entsprechend anpassen. Zu dieser Gruppe gehören `ViewInfo`, `ViewControlPanel` und `ViewPorts`.

5.4.5 DNConsole

Das Befehlszeilen-Werkzeug für das Ausführen von DynamicNodes-Programmen ist die Komponente *DNConsole*. Sie ist als EXE-Datei ebenso direkt ausführbar wie *DNWinApp*, zeigt jedoch keine Windows-Fenster an, sondern startet eine Laufzeitumgebung ohne visuellen Editor und gibt evtl. Benutzer-Nachrichten auf der Konsole aus.

DNConsole ist unabhängig von den Komponenten *DNVisual* und *DNTesting* und sorgt dafür, dass Knoten in der Regel ohne visuelle Erweiterungen ausgeführt werden, so dass bei der Ausführung etwas Arbeitsspeicher gespart wird. Unter Umständen ist auch eine höhere Ausführungsgeschwindigkeit von DynamicNodes-Programmen in *DNConsole* zu erwarten, weil evtl. grafische Ausgaben von Knoten nicht verarbeitet werden.

Eine Übersicht der Befehlszeilenparameter befindet sich in Anhang E - Tabelle 2.

5.5 Qualitätssicherung

Die Qualität des Quellcodes wurde während der Entwicklungszeit durch mehrere Code-Reviews überprüft und anschließend verbessert.

Die Korrektheit der Algorithmen von DynamicNodes wurde nicht mit Hilfe von Unit-Tests, sondern lediglich durch ausführliche Testszenarios überprüft. Dabei wurde die Laufzeitumgebung parallel zu zwei Beispiel-Knotenbibliotheken entwickelt. Die Anforderungen, welche durch die Knotenbibliotheken an die Laufzeitumgebung gestellt wurden, können als Tests verstanden werden. Die Funktionstüchtigkeit der Laufzeitumgebung ist für die, während der Entwicklung entstandenen, Knotenbibliotheken größtenteils bestätigt. Die Beispielknotenbibliotheken wurde so entwickelt, dass sie eine breite Palette von verschiedenen Anforderungen an die Laufzeitumgebung stellt und somit (als Tests betrachtet) eine hohe Testabdeckung erreichen.

Für den Knotenentwickler dient die Testumgebung als Basis für die Überprüfung von Operationsknoten. Dabei kann nicht die Korrektheit der Operation eines Knotens überprüft werden, weil Testfälle in der Testumgebung auf jeden Knotentyp sinnvoll anwendbar sein müssen. Es kann jedoch überprüft werden, ob ein Knotentyp so entwickelt wurde, dass er in der Laufzeitumgebung fehlerfrei ausgeführt werden kann. Dazu werden die zwei Arten von Testfällen, Knotentyp-Test und Knoteninstanz-Test (vergl. 3.5.3), verwendet.

Die Genauigkeit, mit der Knoten auf fehlerhaftes Verhalten, z.B. während der Initialisierung oder während der Ausführung, überprüft werden können, ist von der Anzahl und dem Aufbau der Testfälle abhängig.

In der vorliegenden Version enthält *DNTesting* lediglich eine kleine Anzahl von beispielhaften Testfällen, welche zwar die Kompatibilität von Knoten mit der Laufzeitumgebung für experimentelle Zwecke feststellen können, jedoch keine Sicherheit für einen produktionsähnlichen Einsatz bieten.

Die Benutzeroberfläche wurde während der Entwicklungszeit durch den Autor ständig auf Funktionsfähigkeit, Komfort und Einfachheit überprüft und verbessert.

5.5.1 Portierung auf Linux

Es wurde der Versuch unternommen, DynamicNodes für Mono zu portieren und damit für weitere Plattformen wie Linux und Mac OS X. Dabei wurde deutlich, dass die fehlenden Funktionen in Mono, im Vergleich zum Microsoft .NET-Framework, oft im Bereich Benutzeroberfläche anzutreffen sind.

Es ist durch einige Änderungen am Quelltext gelungen, alle Komponenten von DynamicNodes auf Mono auszuführen. Dabei ist die Benutzeroberfläche aber als instabil und „hakelig“ zu bezeichnen. Das Hilfesystem konnte nicht portiert werden, da das unter Microsoft .NET zur Verfügung stehende WebBrowser-Steuerelement in Mono nicht enthalten ist. Dieses Steuerelement wird zur Anzeige der

HTML-Inhalte der Hilfe und zur automatischen Transformation der Knotenreferenz von XML in HTML verwendet.

Die Komponenten *DNCore* und *DNConsole* laufen nahezu ohne Einschränkungen unter Mono, so dass, vorausgesetzt die Knotenbibliotheken sind ebenfalls Mono-kompatibel, zumindest der Batch-Betrieb von fertigen Programmen unter Linux, Unix oder Mac OS X vorstellbar ist.

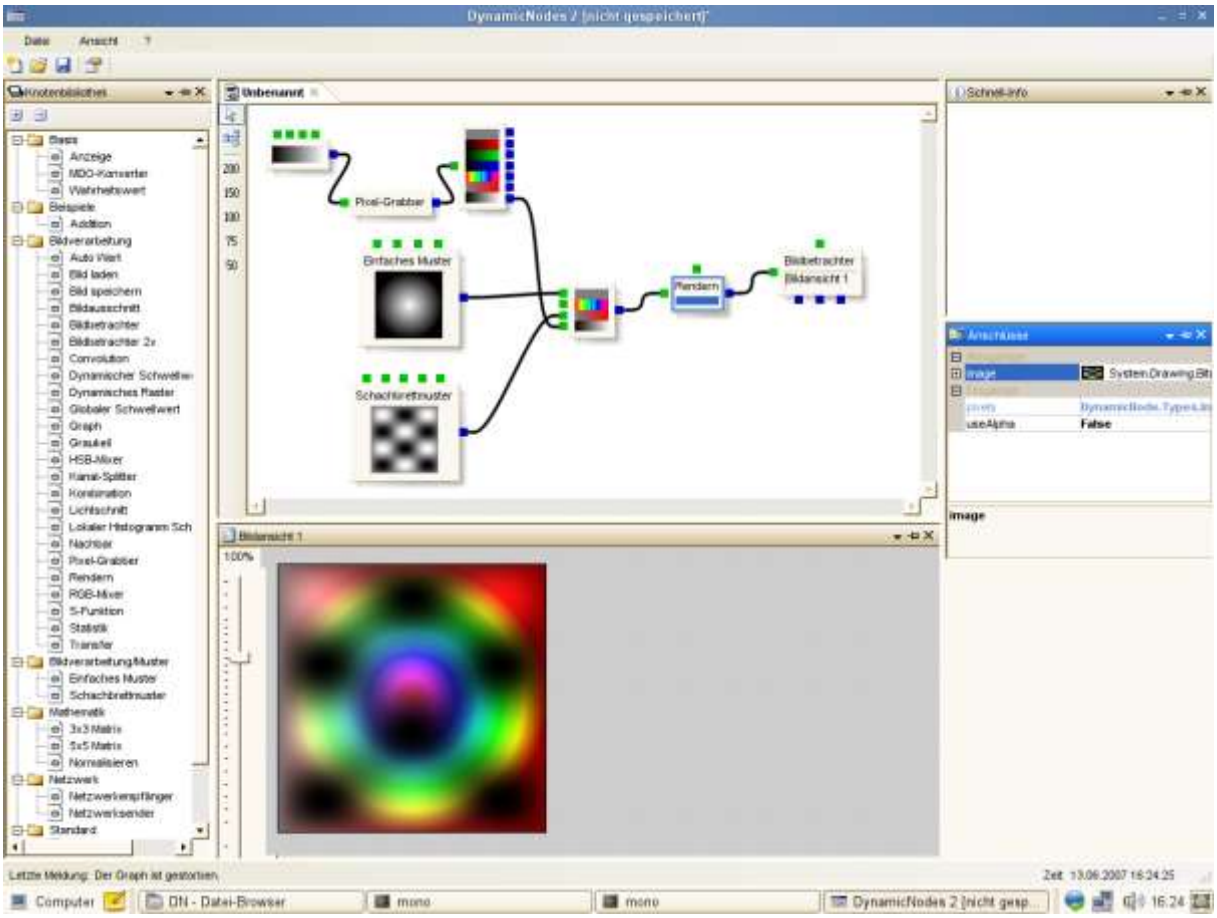
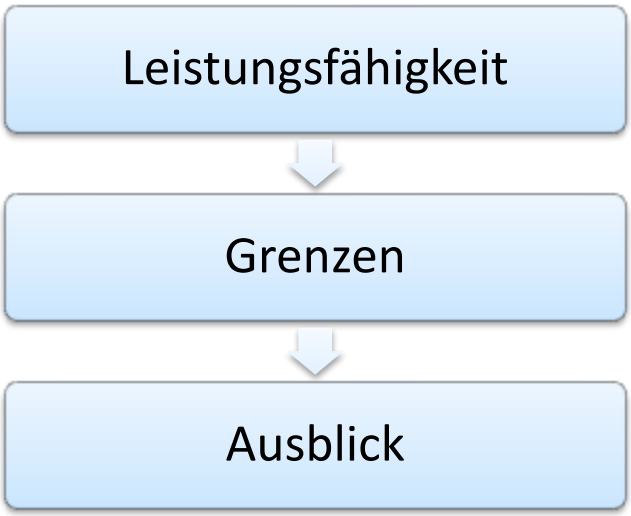


Abb. 69 DynamicNodes unter Ubuntu Linux mit Gnome-Desktop

6 Analyse und Bewertung

Dieses Kapitel betrachtet das Ergebnis der Arbeit und zeigt Leistungsfähigkeit und Grenzen von DynamicNodes auf. Eine Abgrenzung von ähnlicher Software soll die Besonderheiten der vorliegenden Entwicklung hervorheben. Ein abschließender Ausblick gewährt einen kleinen Einblick in die mögliche Zukunft von DynamicNodes.



6.1 Leistungsfähigkeit

In diesem Abschnitt soll kurz die Leistungsfähigkeit von DynamicNodes beleuchtet und bewertet werden. Dabei soll auf die Aspekte Parallelität, Typensicherheit, Flusssteuerung und Komfort eingegangen werden.

6.1.1 Parallelität

DynamicNodes ist in der Lage eine große Anzahl von Operationsknoten in einem oder mehreren Graphen zu verwalten und parallel auszuführen. Dabei kann für jeden Knoten individuell festgelegt werden, ob die Operation parallel oder seriell ausgeführt werden soll. Die Operationen aller Knoten, die parallel ausgeführt werden, werden in Threads abgearbeitet. Die Threads für die Abarbeitung der Operationen werden in einem Pool verwaltet, dessen Größe standardmäßig der Anzahl der Prozessoren entspricht.

Die Zielsetzung, dass DynamicNodes bei der Ausführung von Flussgraphen auf Mehrprozessorsystemen skaliert, ist erreicht worden.

6.1.2 Typensicherheit

Der Knotenentwickler hat die Möglichkeit genau zu definieren, welche .NET-Typen von einem Eingang verarbeitet und von einem Ausgang weitergegeben werden können. Dabei sind je Anschluss mehrere unterstützte Typen möglich. Ein automatischer Mechanismus sorgt dafür, dass zwischen einem Ein- und einem Ausgang der beste gemeinsam unterstützte Datentyp für die Verbindung ausgewählt wird. Dabei hat der Knotenentwickler die Möglichkeit diesen Automatismus durch einen eigenen Algorithmus zu ersetzen.

In der Benutzeroberfläche ist es nicht möglich, zwei inkompatible Anschlüsse miteinander zu verbinden. Durch Drag&Drop-Interaktion kann der Benutzer Verbindungen aufbauen. Dabei wird ihm bei Start der Interaktion an einem Anschluss durch einen Farbwechsel der übrigen Anschlüsse signalisiert, welche von ihnen als Ziel in Frage kommen.

6.1.3 Flusssteuerung

Es gibt verschiedene Ansätze, um eine Flusssteuerung in einem Datenflusssystem zu realisieren. Bei dem Begriff Flusssteuerung handelt es sich hauptsächlich um bedingte Verzweigungen (Alternativen) und um Schleifen. In (Sharp, 1992) werden von Dennis, Rumbaugh, Adams und Kosinski leicht variierenden Vorschläge für Kontrollstrukturen in Form von Knoten gemacht. Dabei wird jeder der Autoren seinem eigenen Datenflussmodell gerecht.

DynamicNodes ist so entworfen worden, dass ein möglichst großes Spektrum an Flusssystemen damit umgesetzt werden kann. Um nachzuweisen, dass DynamicNodes überhaupt in der Lage ist Schleifen und Alternativen abzubilden, wurde eine kleine beispielhafte Knotenbibliothek *NLibTest*

entwickelt, mit deren Knoten die Flusseigenschaften von DynamicNodes untersucht werden können. Alle Knoten arbeiten mit Fließkommazahlen doppelter Genauigkeit als Datentyp.

Folgenden Knoten sind in *NLibTest* enthalten:

- `NGenerator` „Generator“
  Dieser Knoten kann in periodischen Zeitabständen zufällige, ansteigende oder sinusförmige Zahlenwerte produzieren.
- `NComparision` „Vergleich“
  Dieser Knoten vergleicht zwei eingehende Zahlenwerte mit „kleiner“, „gleich“, „größer“ oder „ungleich“ und erzeugt den Wert 1 respektive 0 als Ergebnis.
- `NMerger` „Serialisierer“
  Dieser Knoten gibt Marken, welche an einem der beiden Eingänge eintreffen, sofort an den einzigen Ausgang weiter. Mit diesem Knoten können die Werte zweier Ausgänge zusammengeführt werden.
- `NOperation` „Operation“
  Mit diesem Knoten können zwei Zahlenwerte addiert, subtrahiert oder multipliziert werden.
- `NProcess` „Prozess“
  Dieser Knoten nimmt einen Zahlenwert am Eingang entgegen, führt als Operation eine zeitaufwändige Berechnung durch und gibt den am Eingang enthaltenen Wert unverändert an den Ausgang weiter.
- `NSelection` „Auswahl“
  Mit diesem Knoten kann mit Hilfe einer Bedingung zwischen zwei Verbindungen gewählt werden. Er besitzt drei Eingänge: Den Eingang der Bedingung und zwei Eingänge für die Daten. Ist der Zahlenwert am Bedingungseingang ungleich 0, werden die Marken des ersten Dateneingangs an den Ausgang weitergeleitet, sonst die Marken des zweiten Dateneingangs.
- `NSwitch` „Schalter“
  Dieser Knoten ist das Pendant zu „Auswahl“. Er kann mittels eines Bedingungseingangs die Marken eines Dateneingangs auf zwei Ausgänge verteilen.
- `NTrigger` „Auslöser“
  Dieser Knoten dient der Benutzerinteraktion und kann, ausgelöst mit einem Doppelklick auf die Knotendarstellung, abwechselnd die Werte 0 und 1 produzieren.
- `NDisplay` „Ansicht“
  Dieser Knoten kann die Zahlenwerte, welche an dem einzigen Eingang eintreffen, grafisch darstellen.

Alle Knotenklassen befinden sich im Namensraum `DynamicNode.Lib.Test`. Sie sind der Gruppe „Flussanalyse“ zugeordnet.

Um eine Alternative zu konstruieren, genügt es einen „Schalter“ für die Verzweigung der Eingabewerte und einen „Serialisierer“ für die Zusammenführung der Ergebniswerte zu verwenden. Folgen-

des Beispielprogramm befindet sich im Anhang G - Alternative.dyn (Band 2, S. 451). In grafischer Darstellung sieht das Programm wie in Abb. 70 aus.

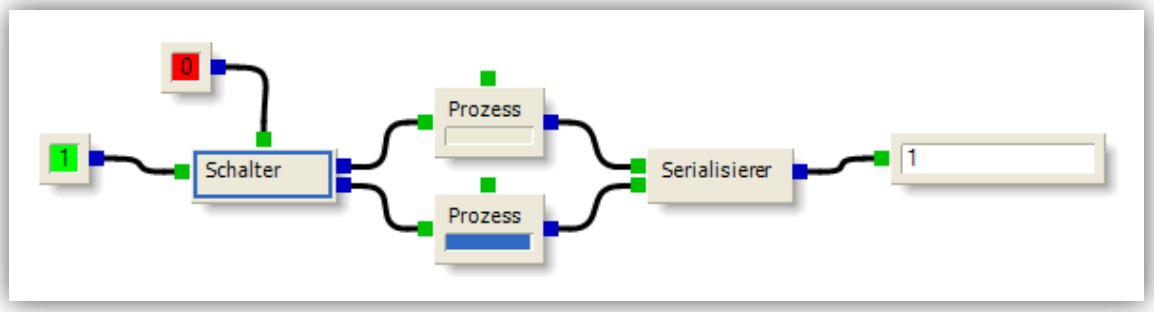


Abb. 70 Beispielprogramm für eine Verzweigung

Eine Schleife ist etwas komplizierter, aber auch ohne Probleme realisierbar. Mit einem „Serialisierer“ werden die Eingabemarken und die Marken der Iterationen vereint. Anschließend wird eine Abbruchbedingung berechnet (hier mit einem „Vergleich“) und diese an einen „Schalter“ geführt. Ist die Abbruchbedingung erfüllt, leitet der „Schalter“ die aktuelle Marke an die „Anzeige“ weiter, sonst wird sie der „Operation“ zugeführt, welche den Zahlenwert in diesem Beispiel inkrementiert. Ein „Prozess“ verlangsamt die Iteration, so dass ein Benutzer den Ablauf verfolgen kann. Das Programm ist im Quelltext im Anhang G - Schleife.dyn (Band 2, S. 454) enthalten. Die grafische Darstellung des Programms zeigt Abb. 71.

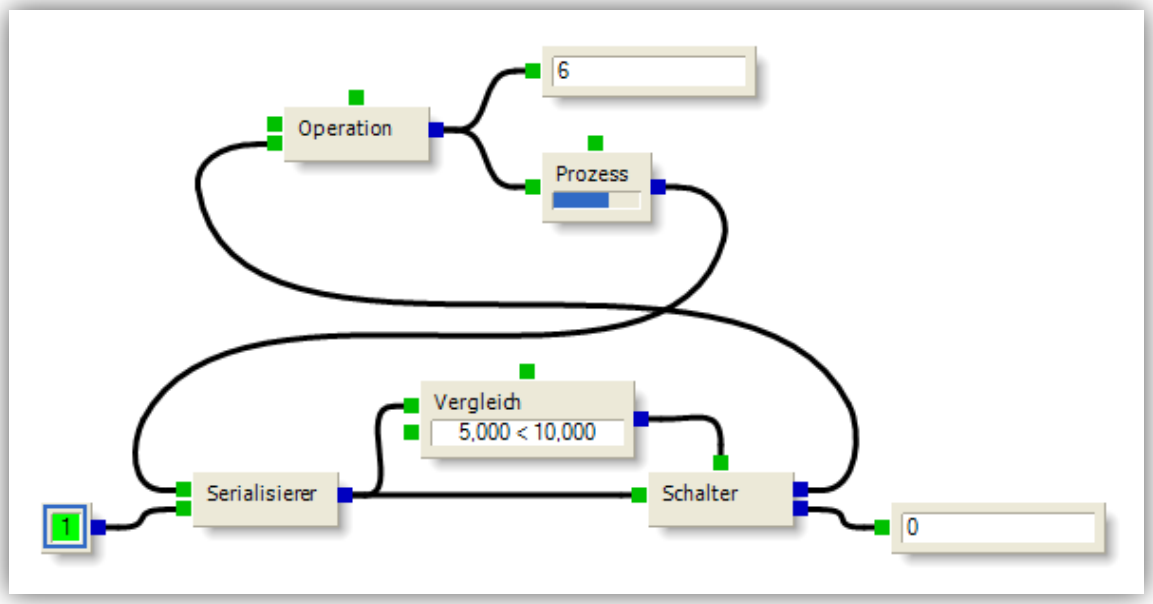


Abb. 71 Beispielprogramm für eine Zählschleife

Beide Beispielprogramme orientieren sich an Vorschlägen, die Rumbaugh mit seiner Notationstechnik macht.

6.1.4 Einsatzmöglichkeiten

In dem jetzigen Entwicklungsstand ist DynamicNodes als ***Experimentierumgebung*** zu betrachten. Es ist gut geeignet, um Aspekte der parallelen Verarbeitung von Daten kennenzulernen.

Ein Einsatz im Hochschulbetrieb ist für verschiedene Bereiche vorstellbar. Prädestiniert ist z.B. die Bildverarbeitung, die Tonverarbeitung, aber auch der Einsatz als Kompositionsplattform für Anwendungskomponenten im Bereich Softwarearchitektur. Voraussetzung dafür ist jeweils die Entwicklung einer Knotenbibliothek, welche die nötigen Operationsknoten zur Verfügung stellt.

Bei geschickter Wahl der Datentypen können Knotentypen leicht anwendungsdomänenübergreifend eingesetzt werden. Es ist ein hoher Grad an Wiederverwendbarkeit möglich. Durch die Fähigkeit von DynamicNodes, Programme mit Hilfe von Aggregation zu verschachteln, ist der Aufbau einer Programmbibliothek auf der Basis von elementaren Operationsknoten denkbar.

Programme deren Funktionsfähigkeit in der Laufzeitumgebung von DynamicNodes ausreichend getestet wurde, können mit Hilfe von *DNConsole* im ***Batchbetrieb*** oder als Dienst ausgeführt werden.

Durch die komfortable Benutzeroberfläche ist es möglich, DynamicNodes als ***Lernplattform*** für eine Anwendungsdomäne einzusetzen. Das Hilfesystem kann leicht um domänenspezifische Inhalte erweitert werden. Mit entsprechenden Knotentypen kann ein Schüler oder Student z.B. den Kontext von Filtern in der Bild- oder Tonverarbeitung kennenlernen.

6.2 Grenzen

Dieser Abschnitt zeigt einige Grenzen von DynamicNodes auf.

DynamicNode ist schlecht geeignet um Flussgraphen zu konstruieren, deren Geschwindigkeit maßgeblich von der Transportgeschwindigkeit der Verbindungen abhängt. Besonders wenn Wertdatentypen zum Einsatz kommen. Denn das automatische In- und Out-Boxing von .NET sorgt zwar für eine korrekte Abbildung von Konstanten und Werttypen auf Verweistypen, benötigt jedoch einiges an Rechen- und Speicherleistung. Schließlich werden die Werte bei jedem Transport über eine Verbindung vom Stack in den Heap verlagert und anschließend wieder zurück.

Die Verbindungen von DynamicNodes unterstützen kein Streaming, d.h. dass ein Knoten während seiner Aktivierung weder mehrere Marken auf einen Ausgang legen kann (es bleibt immer nur die letzte erhalten) noch mehrere Marken von einem Eingang nehmen kann (er hat lediglich Zugriff auf die aktuelle Marke).

Die Vorrausetzung für eine Aktivierung, dass nicht alle Eingänge neue Marken haben müssen, sorgt dafür, dass eine effiziente Aktivierung von Knoten nicht immer möglich ist. Oft tritt der Fall auf, dass ein Knoten mit unvollständigen Eingaben aktiviert wird und später mit den vollständigen Eingabedaten erneut aktiviert wird. Die Rechenressourcen werden dabei nicht optimal genutzt.

Ein *demand-driven*-orientiertes Flussdesign ist mit DynamicNodes ohne weiteres nicht möglich.

Während der Entwicklung ist deutlich geworden, dass vor allem die Verwendung von Threads und die damit verbundenen Risiken für Deadlocks eine hohe Anforderung an das Konzept stellen. Durch

den Entwurf- und Entwicklungsprozess ist nicht nachgewiesen worden, dass die Implementierung der Laufzeitumgebung alle möglichen Ursachen für Deadlocks oder andere Phänomene der Nutzung von gemeinsamem Speicher abfängt bzw. umgeht. Während der Entwicklung sind mehrfach Situationen aufgetreten, bei denen Deadlocks verursacht wurden. In allen Fällen konnte die Ursache behoben werden. Jedoch ist noch ein umfangreicheres Testen notwendig, um von einer stabilen Plattform sprechen zu können.

Die Typensicherheit hängt von der Kompetenz und Sorgfalt des Knotenentwicklers ab, denn die Markenwerte werden nicht auf korrekte Datentypen überprüft. Der Knotenentwickler ist dafür verantwortlich, dass die Anschlüsse das, durch die Kompatibilitätslisten gemachte, Versprechen auch einlösen.

Wenn als Markenwerte echte Referenztypen (ohne Auto-In/Out-Boxing) verwendet werden, muss der Knotenentwickler für die Synchronisation der Zugriffe auf die Objekte sorgen. DynamicNodes bietet keinerlei Unterstützung für den synchronisierten Zugriff auf Referenzdatentypen.

Das Hilfesystem ist, im Gegensatz zur Benutzeroberfläche und den Meta-Daten der Knotentypen, nicht auf Mehrsprachigkeit ausgelegt.

Die Testumgebung enthält nur wenige Testfälle, welche die Kompatibilität von Knotentypen mit der Laufzeitumgebung höchstens ausreichend nachweisen können.

6.3 Ausblick

DynamicNodes zeigt sehr viel Potenzial eine real nutzbare Programmierumgebung werden zu können. Um das Experimentierstadium zu verlassen, ist jedoch eine Weiterentwicklung notwendig. Sinnvoll wäre es die Laufzeitumgebung z.B. mit Unit-Tests stabiler zu machen.

DynamicNodes ist nutzlos ohne gute Knotenbibliotheken. Deswegen ist die Weiterentwicklung der Laufzeitumgebung stark von der Entwicklung solcher Bibliotheken abhängig.

Es gibt eine Menge von elementaren Operationen, welche domänenübergreifend eingesetzt werden können. Dazu zählen primitive mathematische Operationen, Zeichenkettenmanipulationen, Datenzugriffsmöglichkeiten und mehr.

Es wäre sinnvoll die Entwicklung solcher Basisknotenbibliotheken parallel mit der Weiterentwicklung der Laufzeitumgebung zu betreiben, da beide Entwicklungswege voneinander profitieren können.

Neben den eben erwähnten Basisoperationen gibt es eine Reihe weiterer Knotentypen, die anwendungsübergreifend eingesetzt werden können und die zu den Standardbibliotheken von DynamicNodes zählen sollten. Dazu gehört ein Knoten, der die Ausführung eines Quelltextes als Operation ermöglicht. Also eine Form von skriptbaren Knoten. Für die Umsetzung können die in .NET integrierten Compiler für C#, VB und JScript verwendet werden. Ein weiterer Knoten könnte ein Adapter für

C-DLLs sein, mit dem der Aufruf von exportierten C-Funktionen in DLLs als Knotenoperation möglich ist. Auch Knoten für die Flusssteuerung wären sinnvoll. Wobei diese Knoten jede Art von Datentypen weiterleiten müssten, dabei aber die Typensicherheit waren sollten.

Abschließend lässt sich sagen, dass eine Weiterentwicklung von DynamicNodes unbedingt empfehlenswert ist. Der Autor wird nach Abschluss dieser Arbeit die Möglichkeiten für eine Weiterentwicklung privat, im Rahmen einer Firma oder in einer OpenSource-Community untersuchen.

7 Anhang

A. Abbildungsverzeichnis

B. Literaturverzeichnis

Adams, D. A. (1970). *A model for parallel computations.*

Balzert, H. (2005). *Lehrbuch, Grundlagen der Informatik* (2. Auflage Ausg.). München: Elsevier GmbH, Spektrum Akademischer Verlag.

Britcher, R. N., & Craig, J. J. (1986). *Using Modern Design Practices to Upgrade Aging Software Systems.* IEEE Software.

Creutzburg, R. (2003). *Skript zur Vorlesung "Algorithmen und Datenstrukturen".*

Dennis, J. B. (1974). *First Version of a Data Flow Language.* Springer Verlag.

Digital Mars. (kein Datum). *Garbage Collection - D Programming Language - Digital Mars*. Abgerufen am 04. 06 2007 von Digital Mars: http://www.digitalmars.com/d/garbage.html

ECMA - Mono. (kein Datum). Abgerufen am 5. Juli 2007 von Mono: http://mono-project.com/ECMA

Ernst, H. (2000). *Grundlagen und Konzepte der Informatik* (2. Auflage Ausg.). Braunschweig/Wiesbaden: Vieweg.

Kosinski, P. R. (1973). *A data flow programming language for operating systems.*

Lima, I. G., Mundy, D., & Treleaven, P. C. (1983). *Decentralised Control Flow Programming, Information Processing 83.* North-Holland: Elsevier Science Publishers B. V.

MainPage - Mono. (kein Datum). Abgerufen am 5. Juli 2007 von Mono: http://mono-project.com/Main_Page

Microsoft. (2007). *Assemblies*. Abgerufen am 4. Juli 2007 von MSDN Microsoft Developer Network: http://msdn.microsoft.com/library/deu/default.asp?url=/library/DEU/cpguide/html/cpconassemblieoverview.asp

Microsoft Deutschland. (24. Juni 2004). *Allgemeine Einführung: .NET und .NET-Framework*. Abgerufen am 4. Juli 2007 von MSDN Deutschland: http://www.microsoft.com/germany/msdn/library/net/AllgemeineEinfuehrungNETUndNETFramework.mspx

Mündemann, F. W. (1989). *Konzepte zur Verwaltung n-dimensionaler Felder als Operanden in Datenfluß-synchronisierten Programmen.* München, Univ. d. Bundeswehr.

Petri, C. A. (1962). *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik.

Reisig, W. (2006). *Petri-Netze - Eine neue Einführung.* Springer-Verlag.

Rumbaugh, J. E. (1977). *A data flow multiprozessor.*

Schäpers, A., Huttary, R., & Bremes, D. (2002). *C#, Windows- und Web-Programmierung mit Visual Studio .NET.* München, Bayern, Germany: Markt+Technik.

Sharp, J. A. (1992). *Data flow computing: Theory and Practice.* Ablex Publishing Corporation.

Treleaven, P. C., Brownbridge, D. R., & Hopkins, R. P. (1982). *Data-Driven and Demand-Driven Computer Architecture.* ACM Computer Surveys.

Volkmann, L. (1996). *Fundamente der Graphentheorie.* Springer-Verlag.

Waldschmidt, K., Bode, A., Brüning, U., Chapman, B. M., Dal Chin, M., Händler, W., et al. (1995). *Parallelrechner: Architekturen - Systeme - Werkzeuge.* (K. Waldschmidt, Hrsg.) Stuttgard: Teubner.

C. Abgrenzung

Die Idee, eine visuelle Programmierumgebung für Flusssysteme zu entwickeln, ist nicht neu in der Softwaregeschichte. An dieser Stelle sollen zwei bekannte Produkte erwähnt und einige Unterschiede zu DynamicNodes erläutert werden.

MathWorks Simulink

http://www.mathworks.com/products/simulink/

Simulink ist eine modellbasierte Entwicklungsplattform, integriert in MATLAB. Sie bietet einen grafischen Editor und eine große Zahl von Bibliotheken. Die Erweiterung um eigene Funktionsblöcke ist möglich. Im Hinblick auf Funktionsfähigkeit und Produktreife kann Simulink alles was DynamicNodes bietet und noch wesentlich mehr. Jedoch hat Simulink eines nicht, eine einfache Benutzeroberfläche für Schüler oder Studenten. Vielmehr hat diese Software den Anspruch als Werkzeug in Wissenschaft und technischer Produktentwicklung zu dienen. Die damit einhergehende Komplexität, fordert ihren Tribut in Form einer umfangreichen und notwendigen Einarbeitung.

Simulink ist ein kommerzielles Werkzeug, welches nach einem Baukastenprinzip um kostenpflichtige AddOns erweitert werden kann.

vvvv

http://vvvv.org/

vvvv ist eine visuelle Entwicklungsumgebung für Multimediasysteme. Ein intuitiver Editor für eine große Anzahl von verschiedenen Knoten ermöglicht z.B. in wenigen Schritten das „Zusammenklicken“ einer DirectX-Animation, nach Belieben auch unter Verwendung von Videos. vvvv ist ein ausgereiftes Werkzeug zur Erstellung von interaktiven Multimediaproduktionen. Die Erweiterung um eigene Knoten ist nicht ohne weiteres möglich. Die Software kann für nicht kommerzielle Zwecke kostenlos verwendet werden.

Kurzer Vergleich

Der größte Unterschied zwischen DynamicNodes und Simulink ist, dass DynamicNodes eine einfache und intuitive Oberfläche bietet, um Einsteigern so wenig Hürden wie möglich in den Weg zu legen, während Simulink für ausgebildete Fachkräfte und Wissenschaftler entwickelt wurde.

vvvv besitzt eine starke Ausrichtung auf den Multimedia-Bereich und ist nicht ganz so universell einsetzbar wie DynamicNodes. Jeder Programmierer, der eine .NET-Sprache beherrscht, kann mit einem minimalen Aufwand eigene Knoten und Knotenbibliotheken entwickeln, um die Funktionen von DynamicNodes zu erweitern. Das ist für vvvv nicht möglich.

D. Diagramme

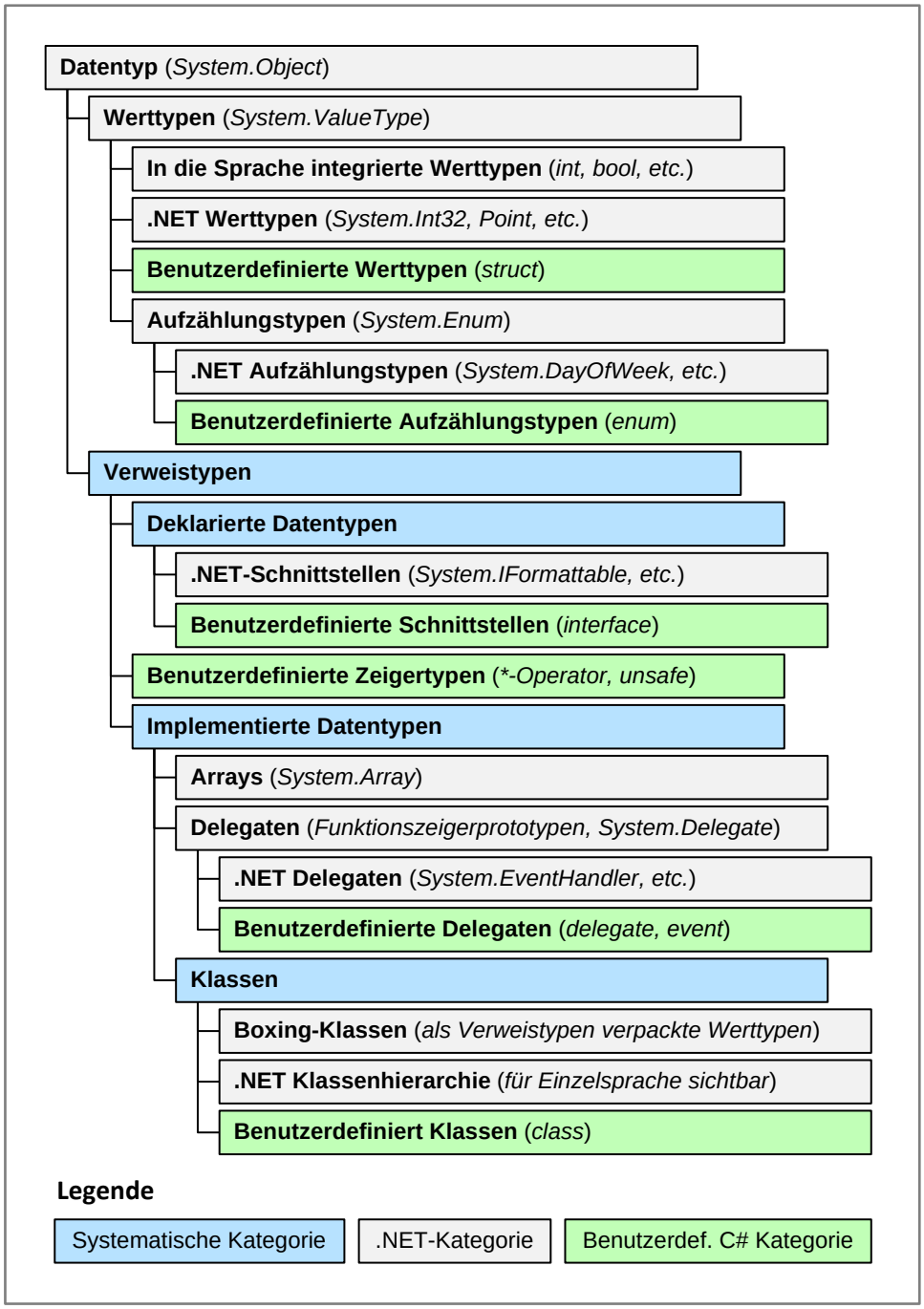


Diagramm 1 Common Type System

Typen-Diagramme

Dieser Abschnitt enthält einige Klassenübersichten (für die verschiedenen Komponenten) und eine größere Anzahl Klassendiagramme. Für das Verständnis der Klassendiagramme ist sicherlich das Beherrschen einer objektorientierten Programmiersprache Voraussetzung. Der interessierte Leser mit entsprechenden Vorkenntnissen findet jedoch in den Klassendiagrammen einige Details, dessen Erwähnung den Rahmen des Hauptteils gesprengt hätte.

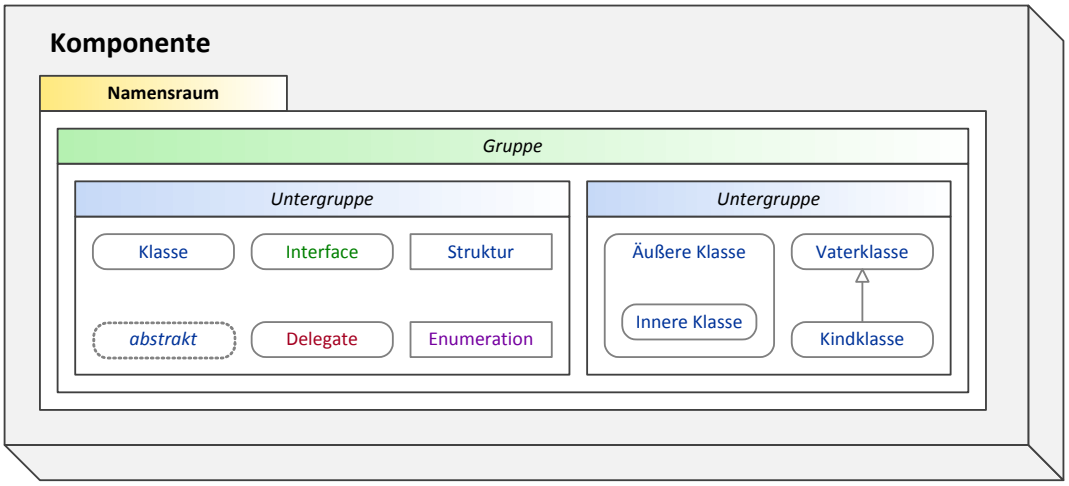


Diagramm 2 Beispiel für eine Klassenübersicht mit erläuternder Beschriftung

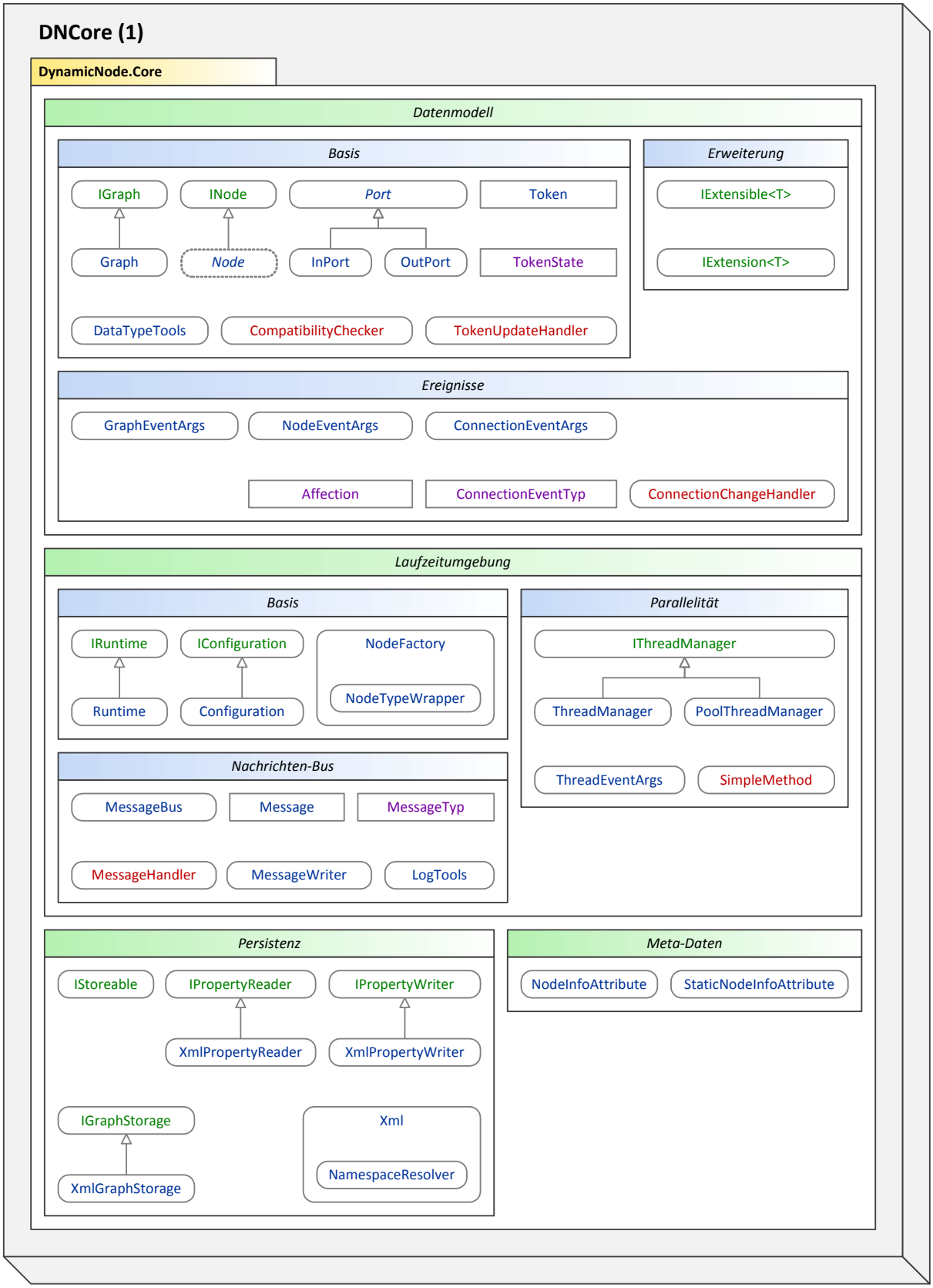


Diagramm 3 Klassenübersicht für DNCore Teil 1

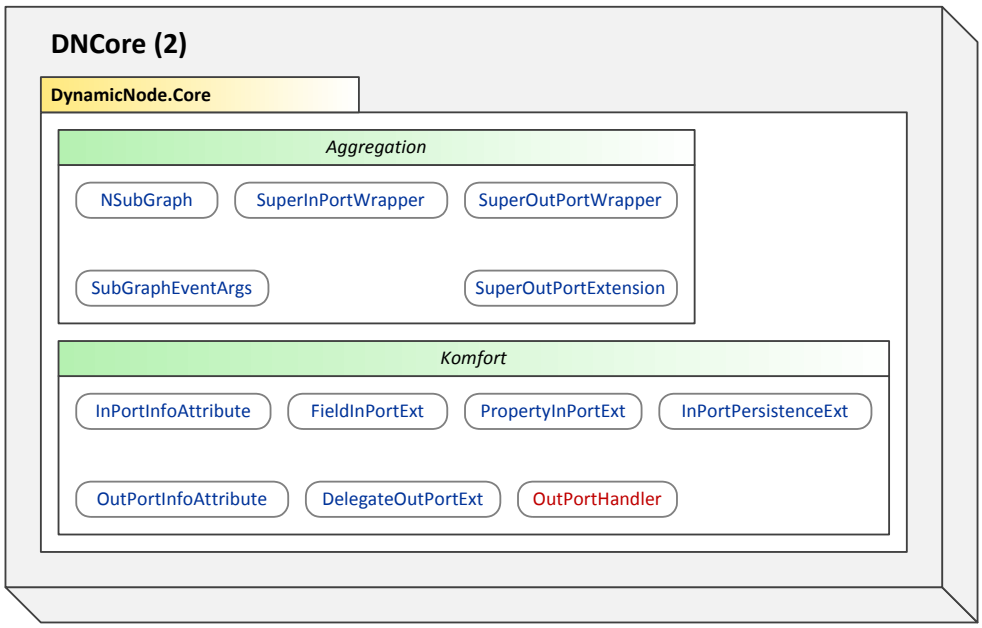


Diagramm 4 Klassenübersicht für DNCore Teil 2

Diagramm 5 DynamicNode.Core.Token

Diagramm 6 DynamicNode.Core.TokenState

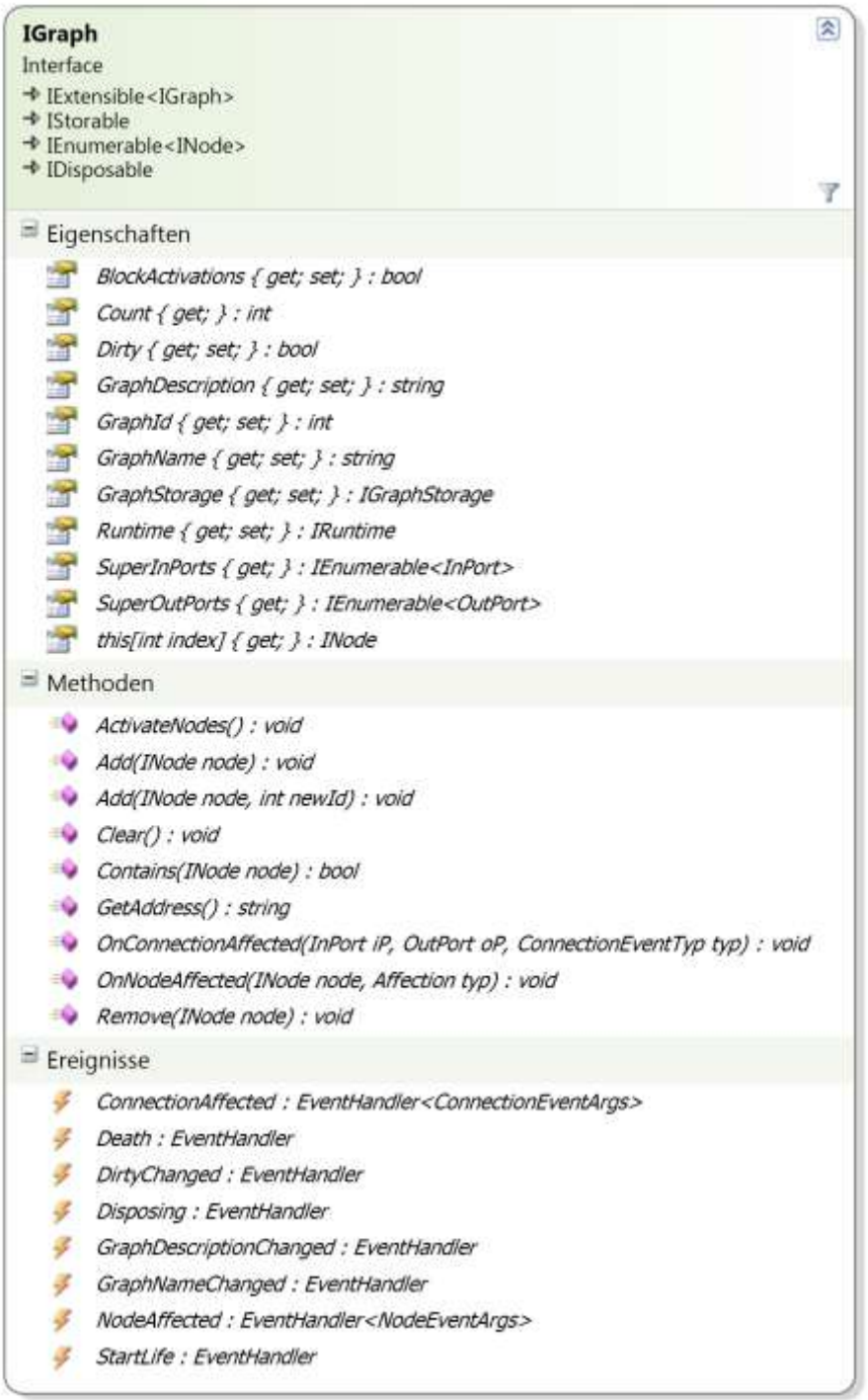


Diagramm 7 DynamicNode.Core.IGraph

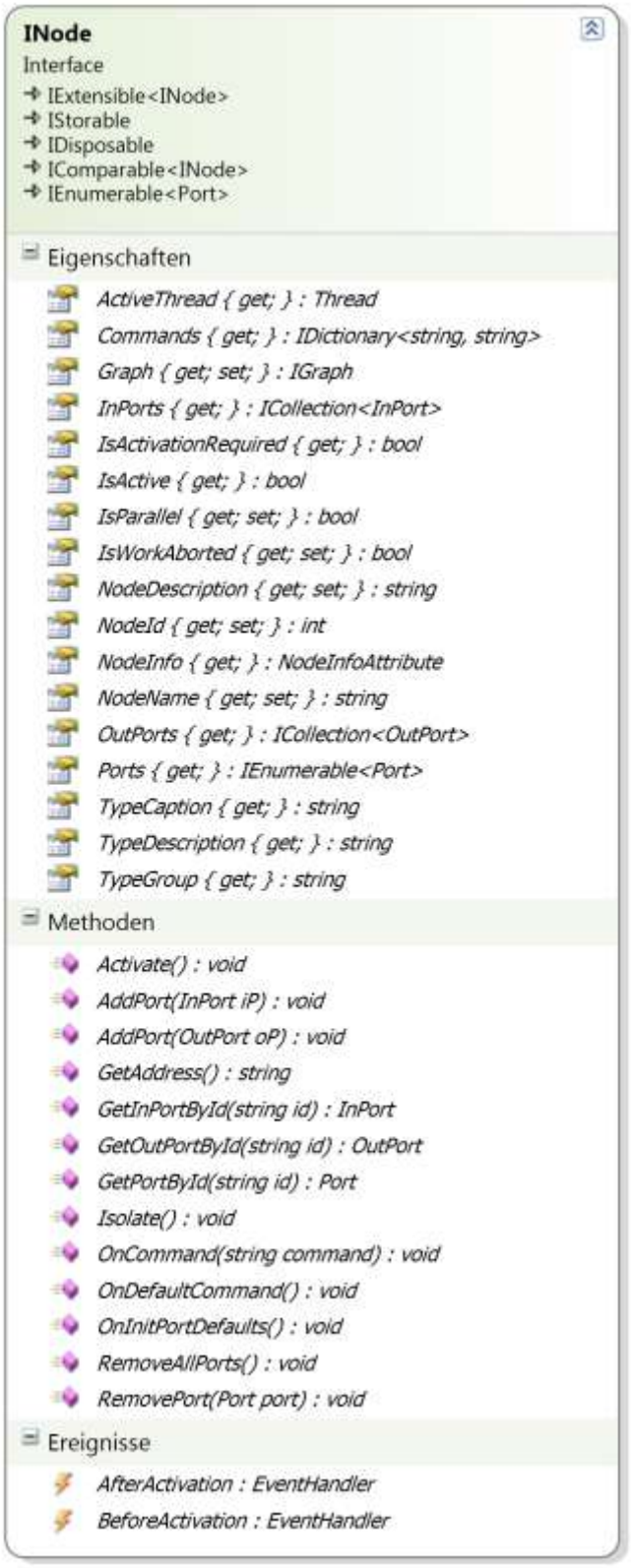


Diagramm 8 DynamicNode.Core.INode

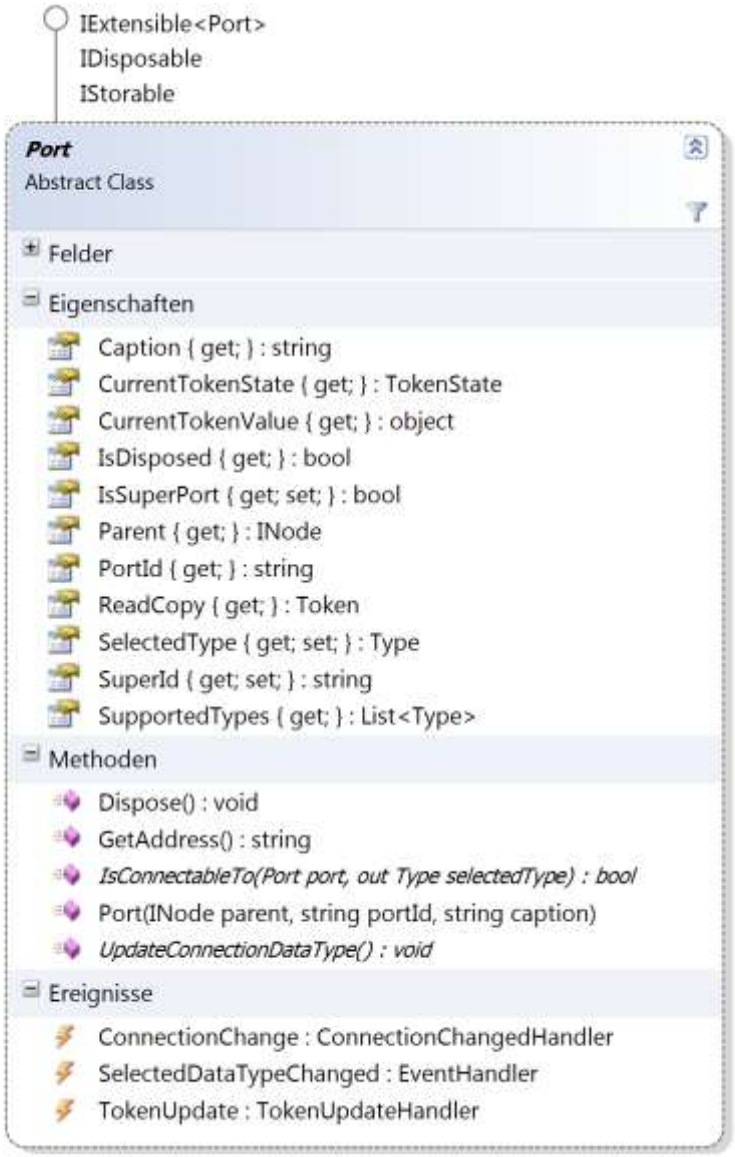


Diagramm 9 DynamicNode.Core.Port

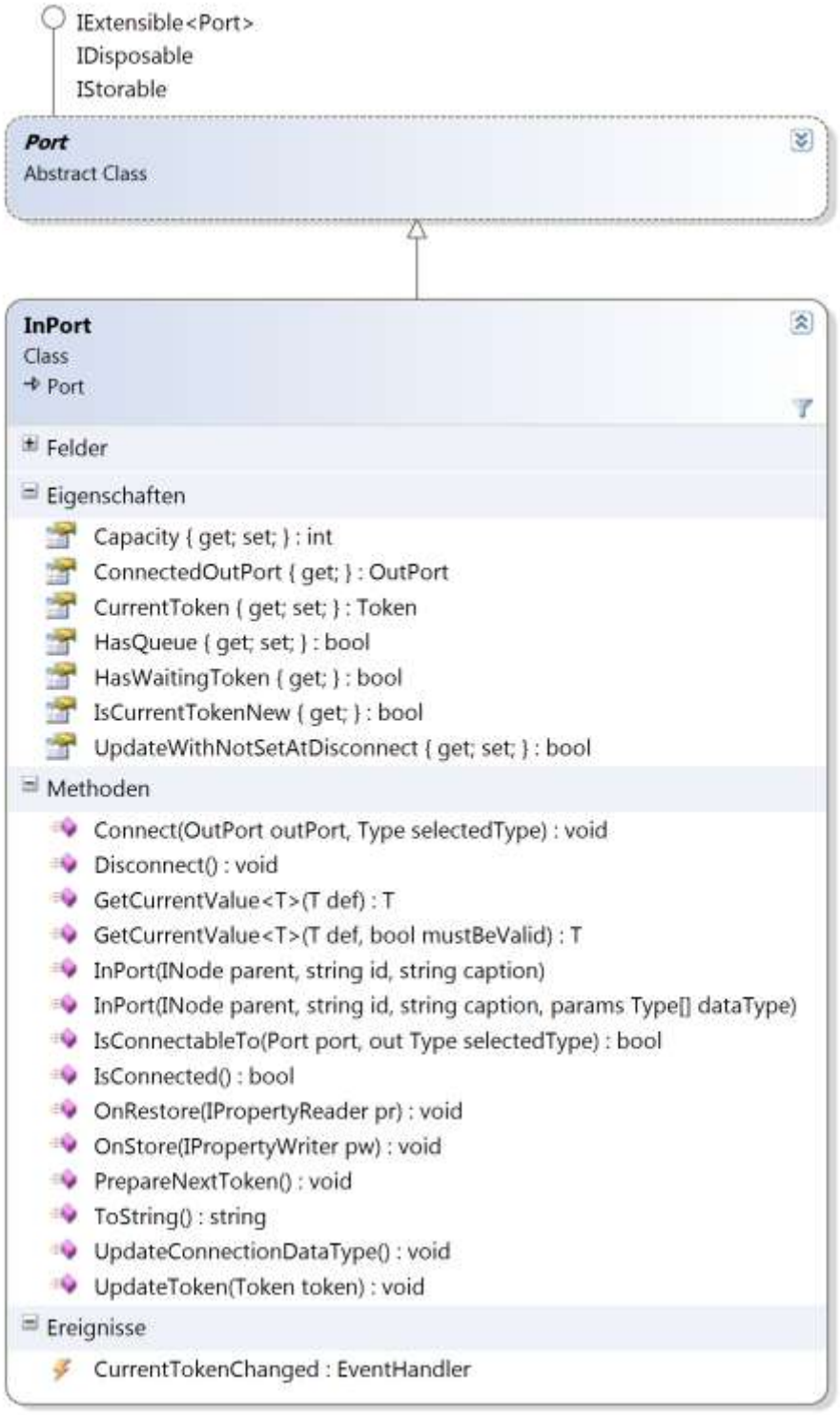


Diagramm 10 DynamicNode.Core.InPort

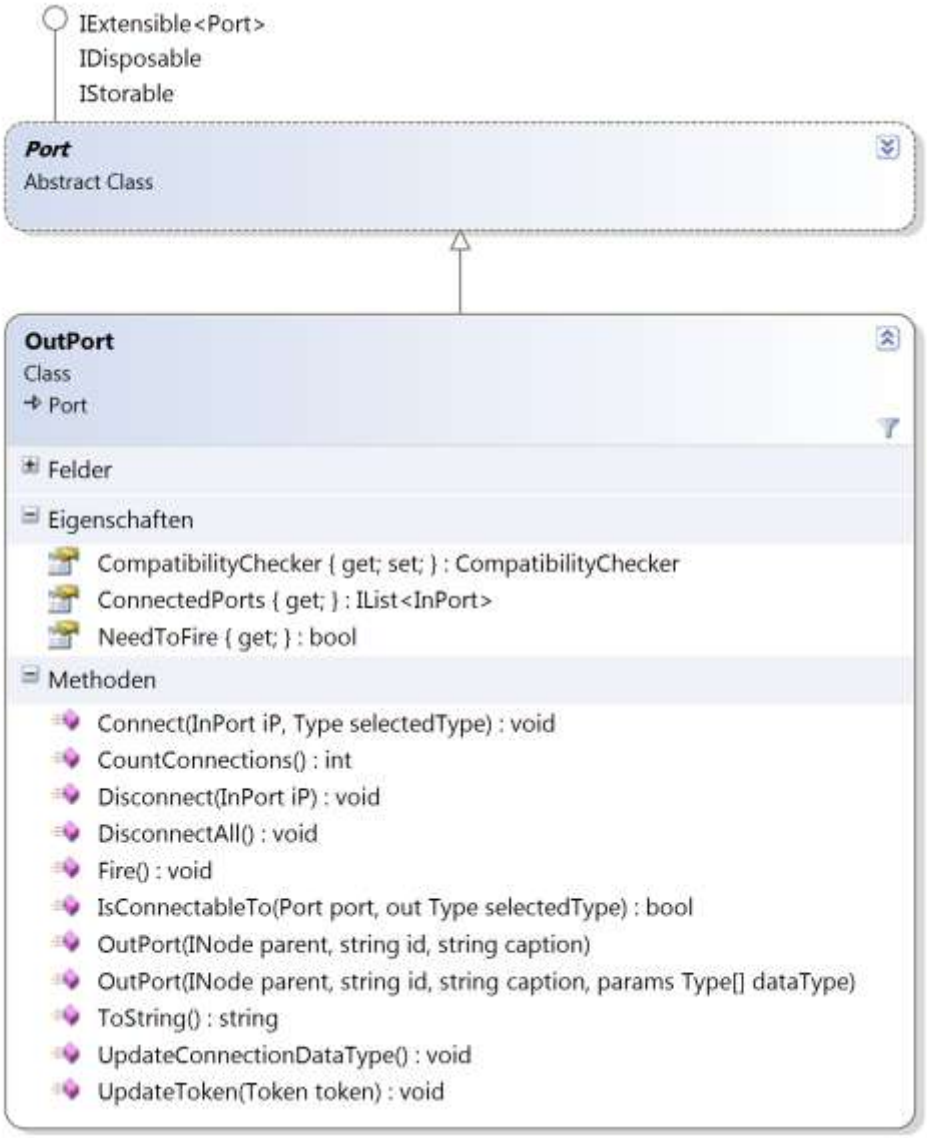


Diagramm 11 DynamicNode.Core.OutPort

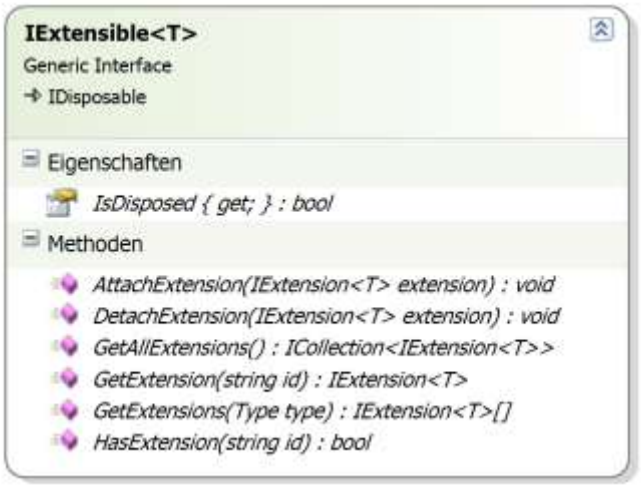


Diagramm 12 DynamicNode.Core.IExtensible<T>

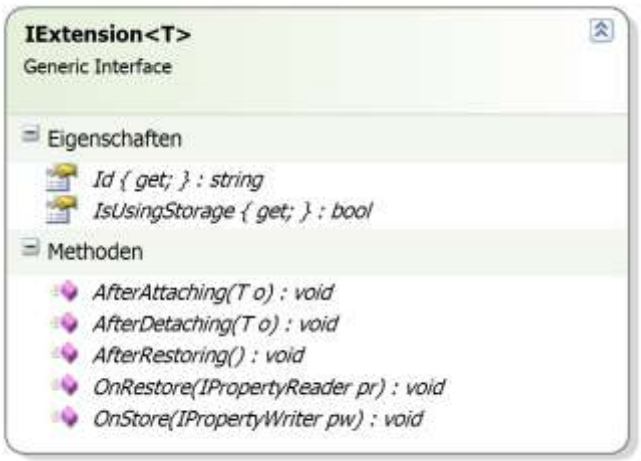


Diagramm 13 DynamicNode.Core.IExtension<T>

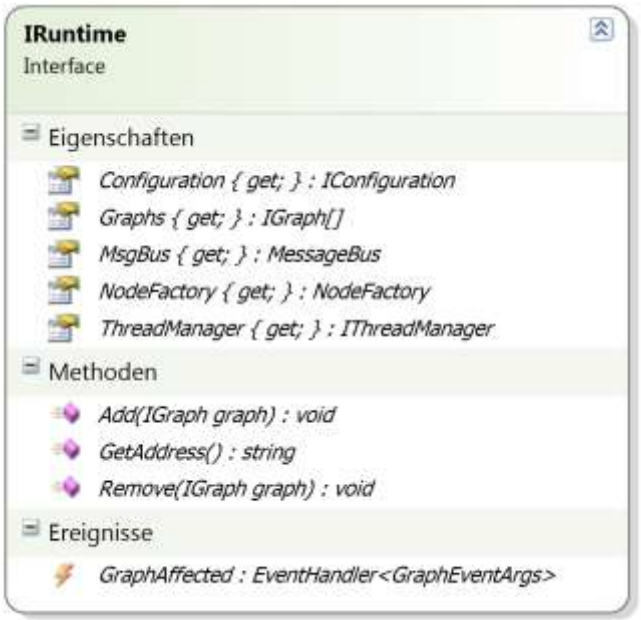


Diagramm 14 DynamicNode.Core.IRuntime

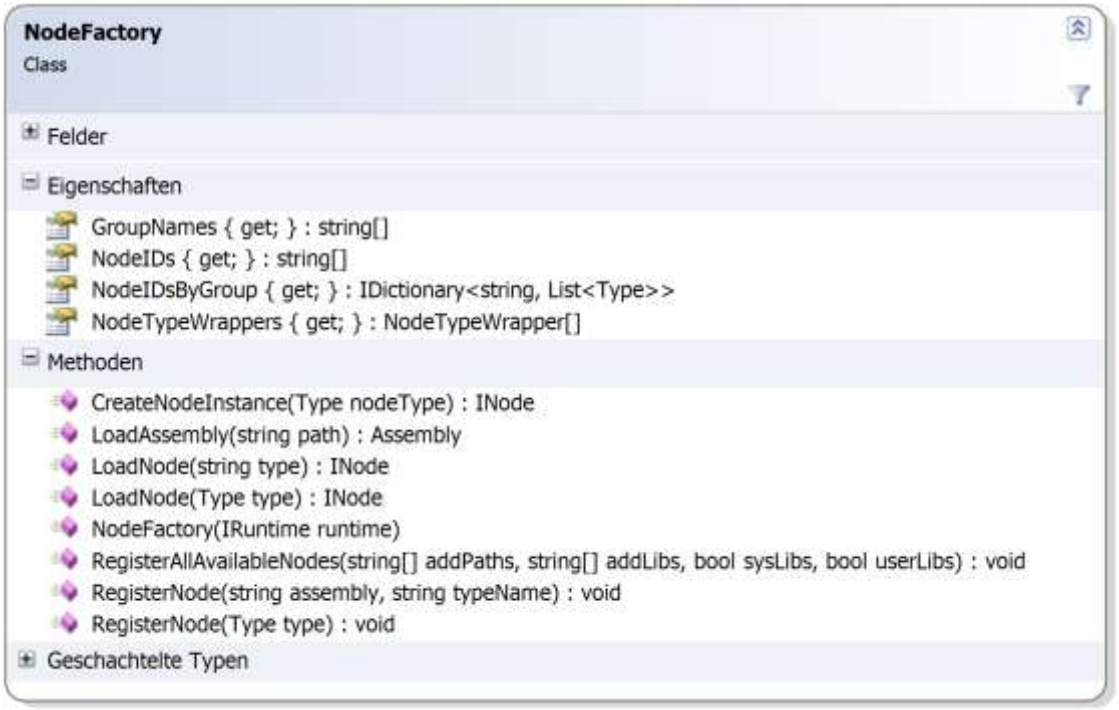


Diagramm 15 DynamicNode.Core.NodeFactory

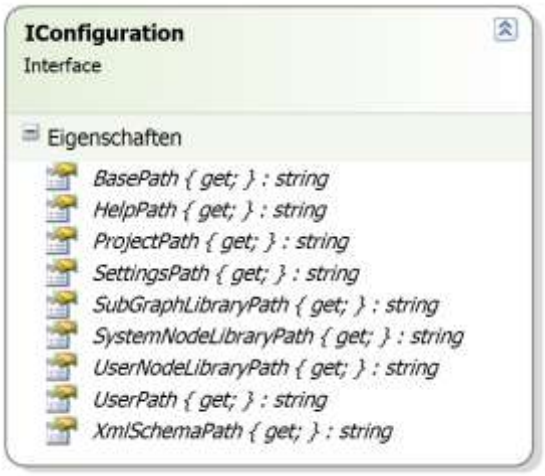


Diagramm 16 DynamicNode.Core.IConfiguration

Diagramm 17 DynamicNode.Core.IThreadManager

Diagramm 18 DynamicNode.Core.MessageBus

Diagramm 19 DynamicNode.Core.Message

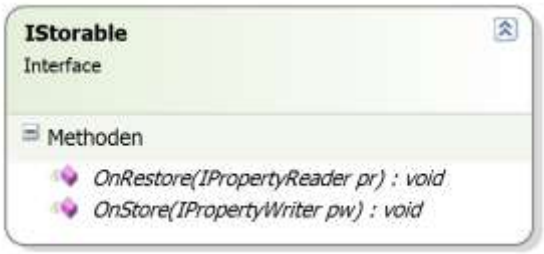


Diagramm 20 DynamicNode.Core.IStoreable

Diagramm 21 DynamicNode.Core.IPropertyReader

Diagramm 22 DynamicNode.Core.IPropertyWriter

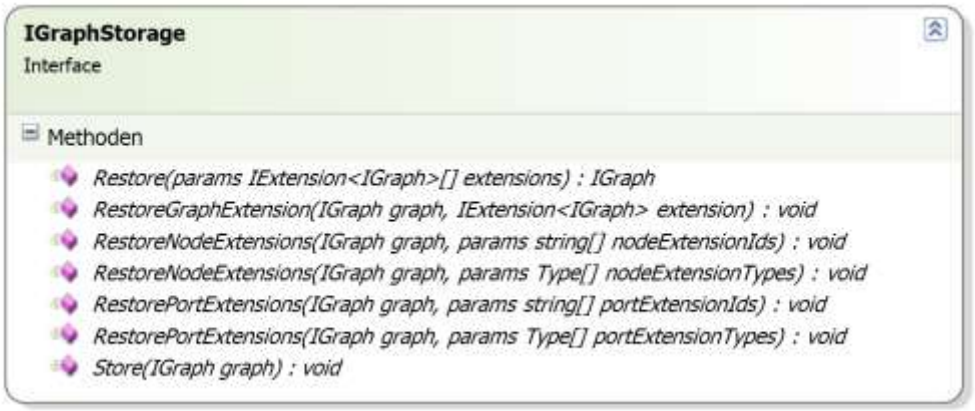


Diagramm 23 DynamicNode.Core.IGraphStorage

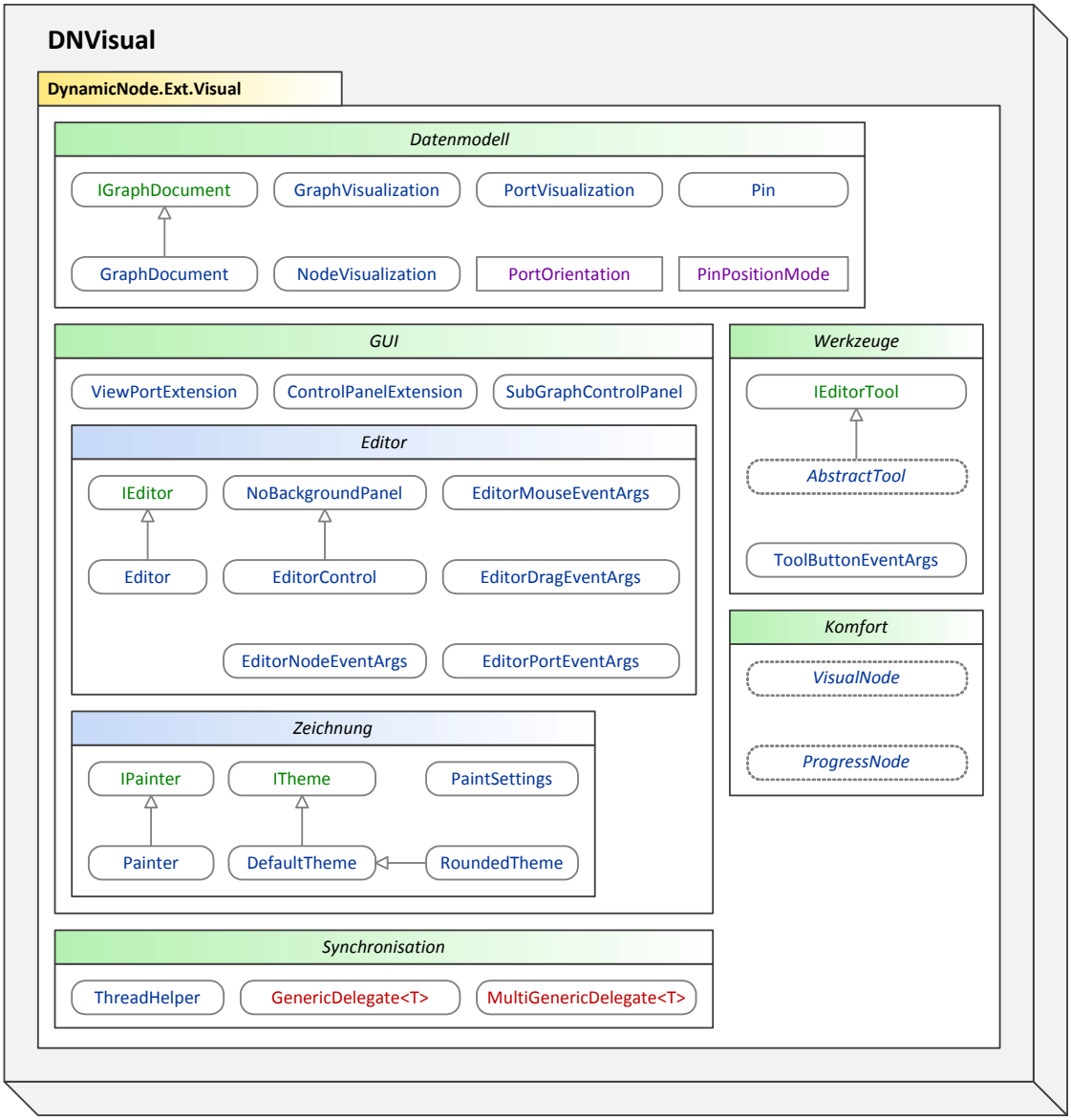


Diagramm 24 Klassenübersicht für DNVisual

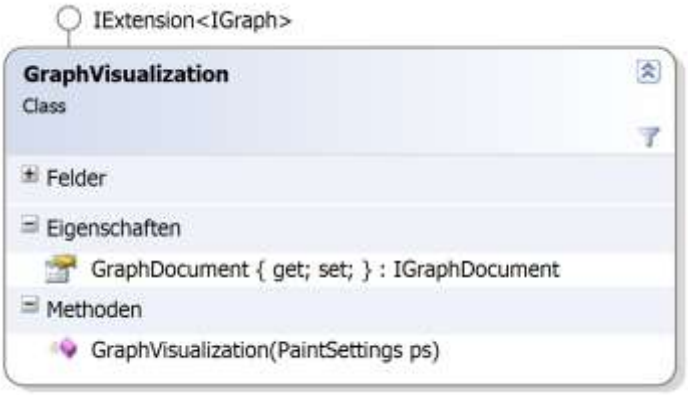


Diagramm 25 DynamicNode.Ext.Visual.GraphVisualization

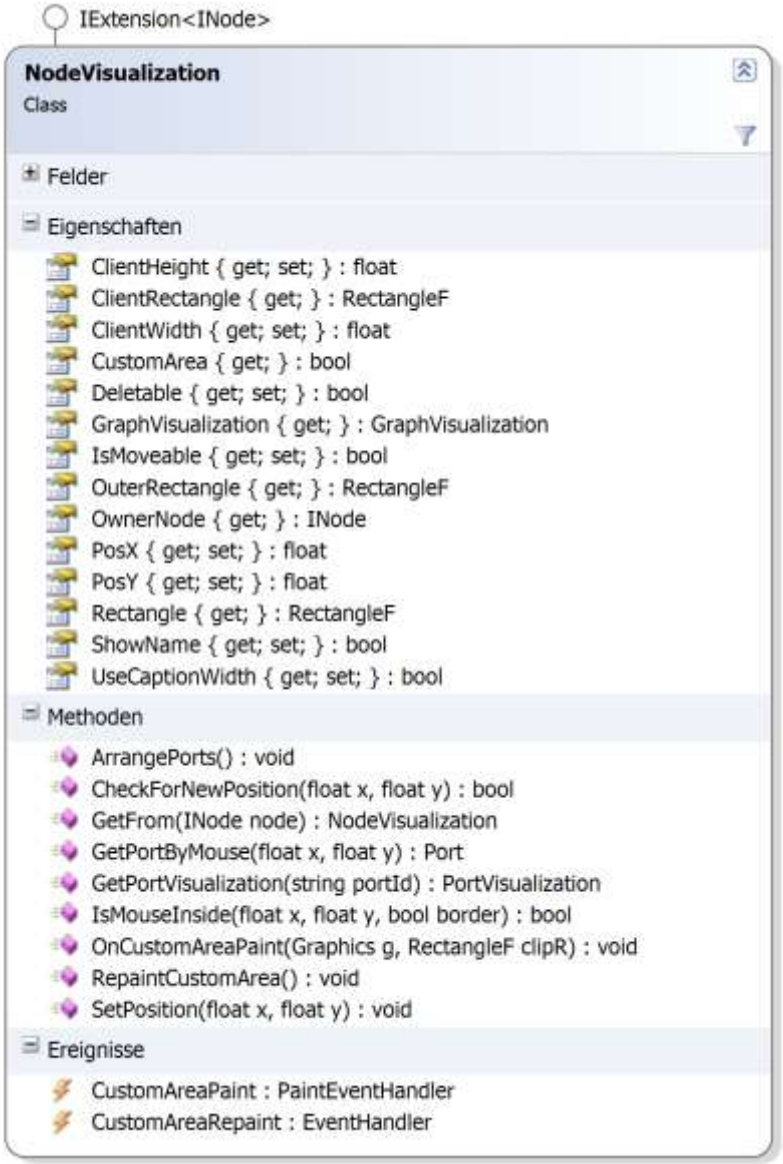


Diagramm 26 DynamicNode.Ext.Visual.NodeVisualization

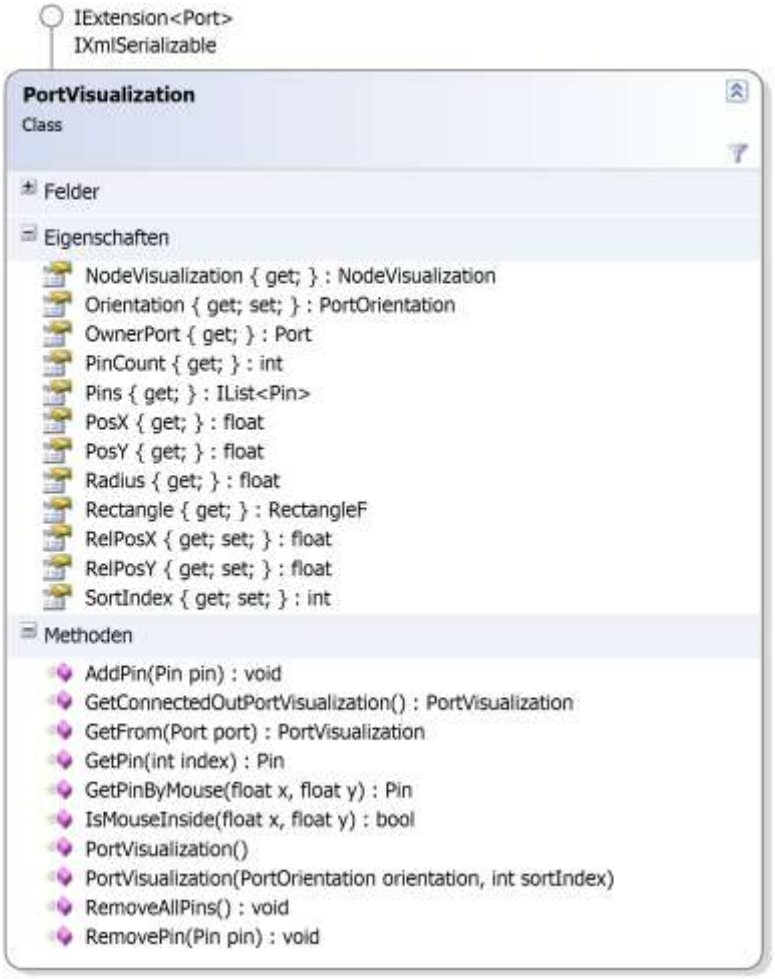


Diagramm 27 DynamicNode.Ext.Visual.PortVisualization

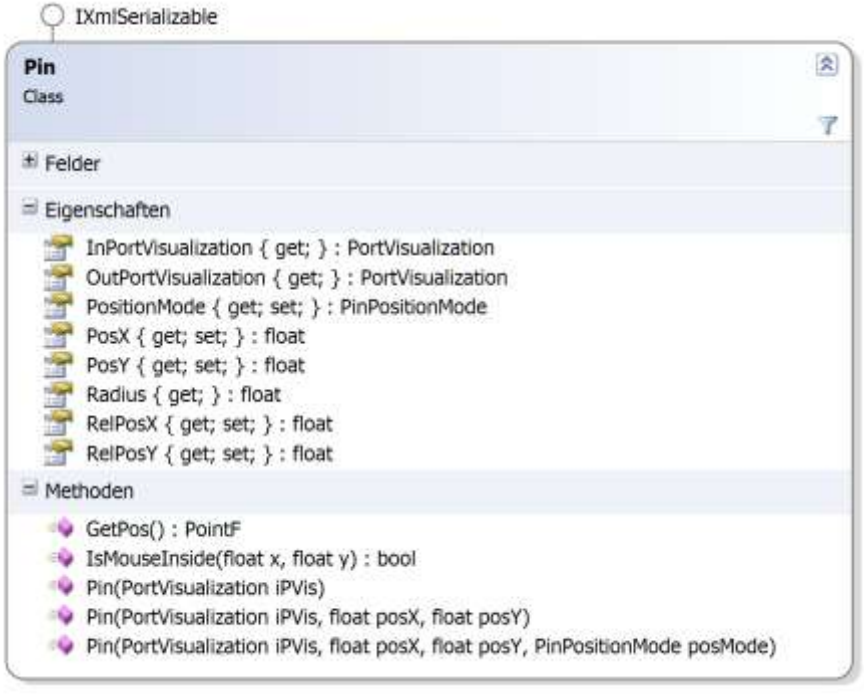


Diagramm 28 DynamicNode.Ext.Visual.Pin

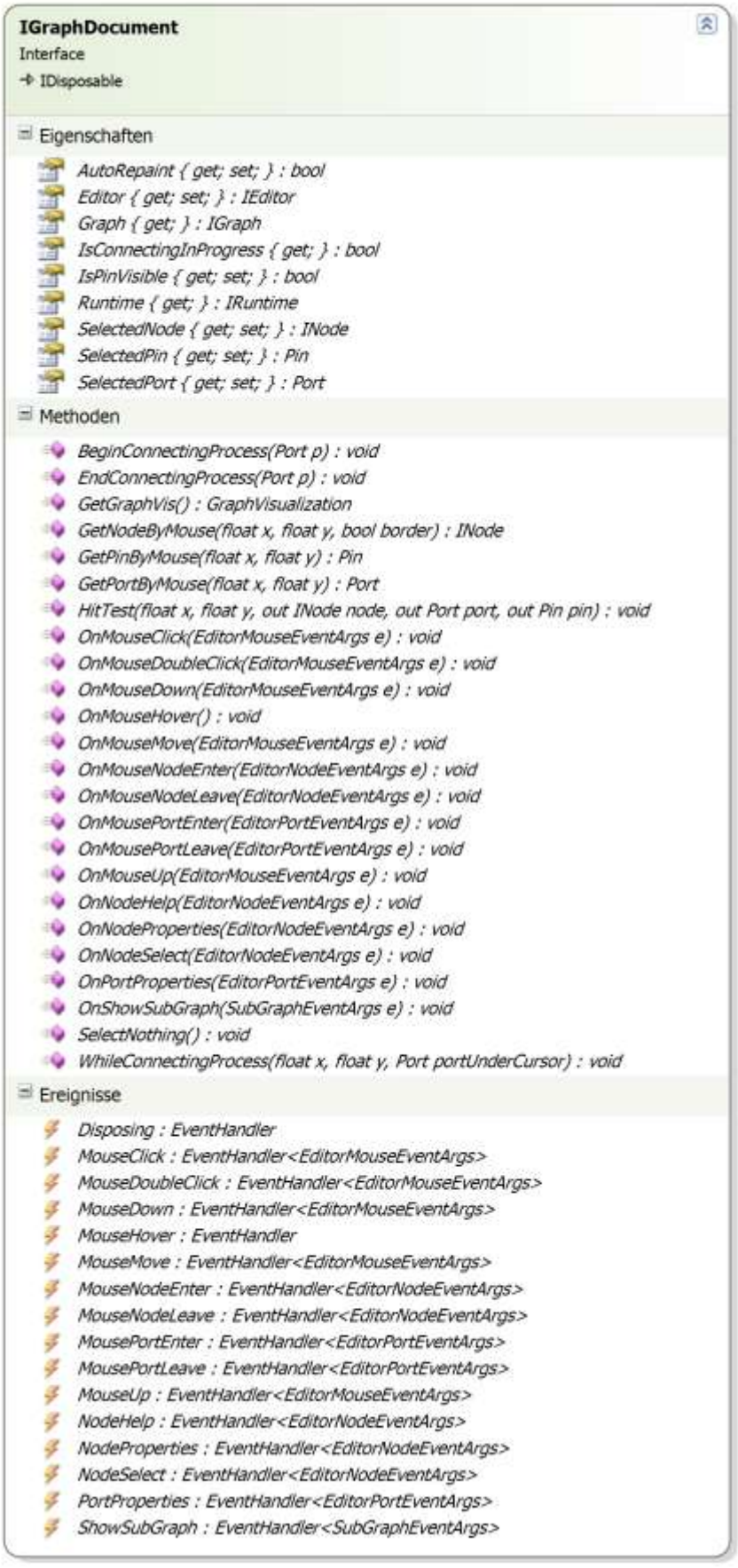


Diagramm 29 DynamicNode.Ext.Visual.IGraphDocument

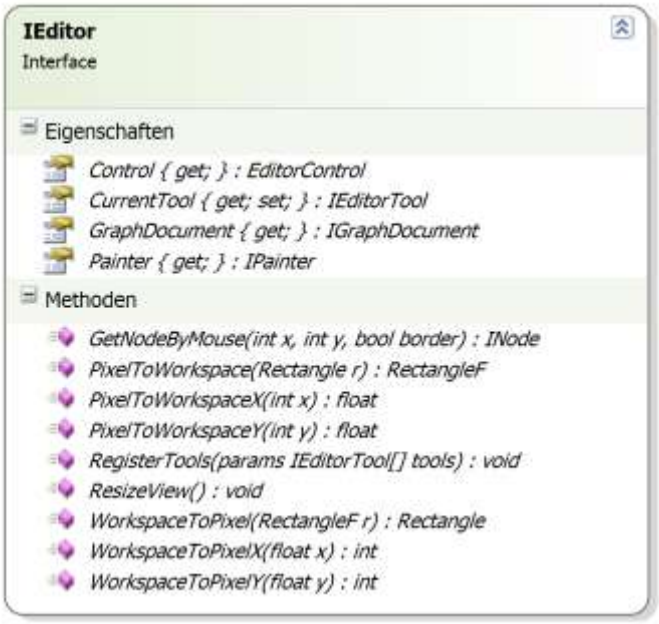


Diagramm 30 DynamicNode.Ext.Visual.IEditor

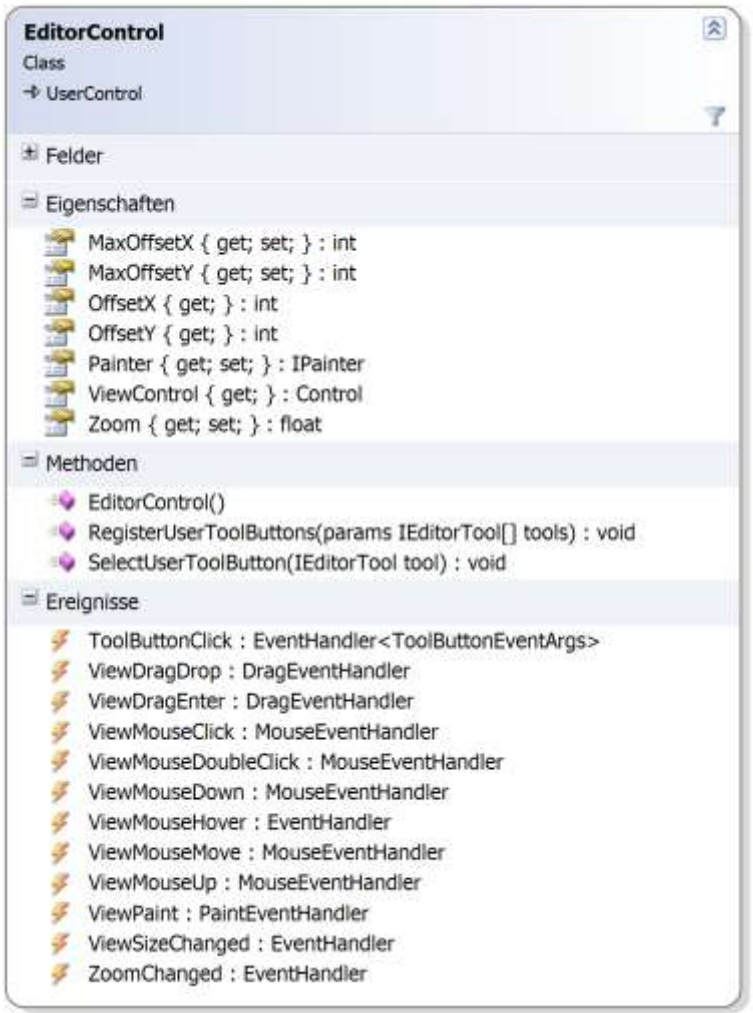


Diagramm 31 DynamicNode.Ext.Visual.EditorControl

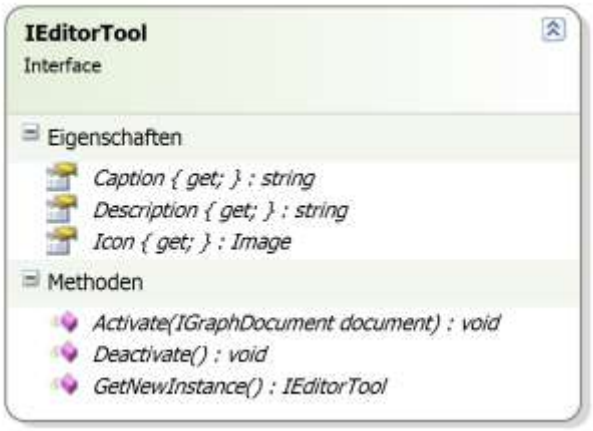


Diagramm 32 DynamicNode.Ext.Visual.IEditorTool

Diagramm 33 DynamicNode.Ext.Visual.IPainter

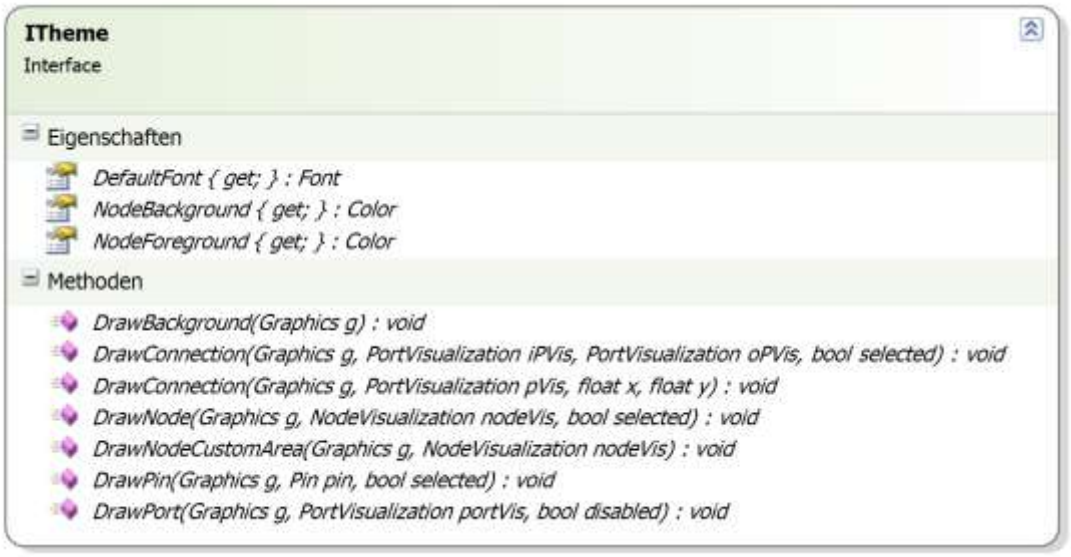


Diagramm 34 DynamicNode.Ext.Visual.ITheme

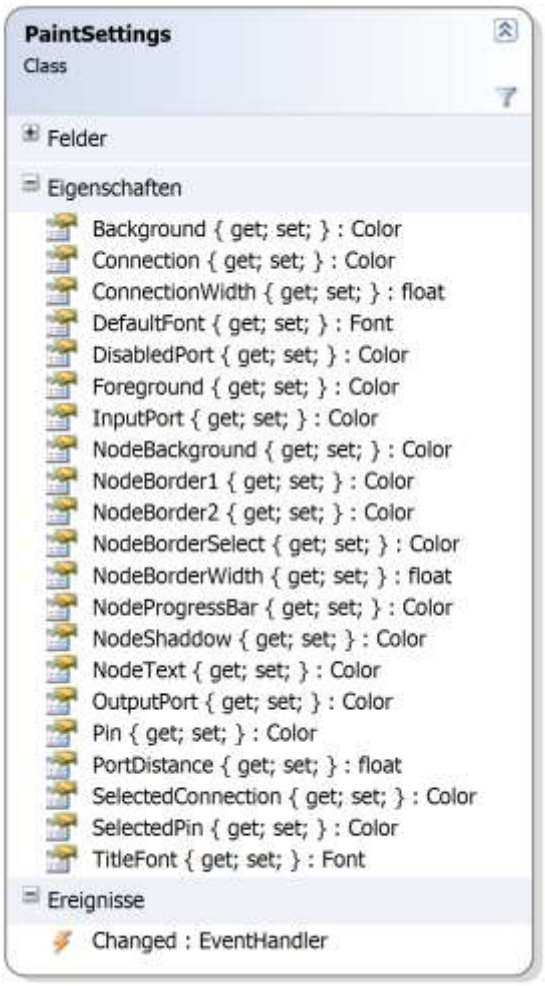


Diagramm 35 DynamicNode.Ext.Visual.PaintSettings

Diagramm 36 DynamicNode.Ext.Visual.ControlPanelExtension

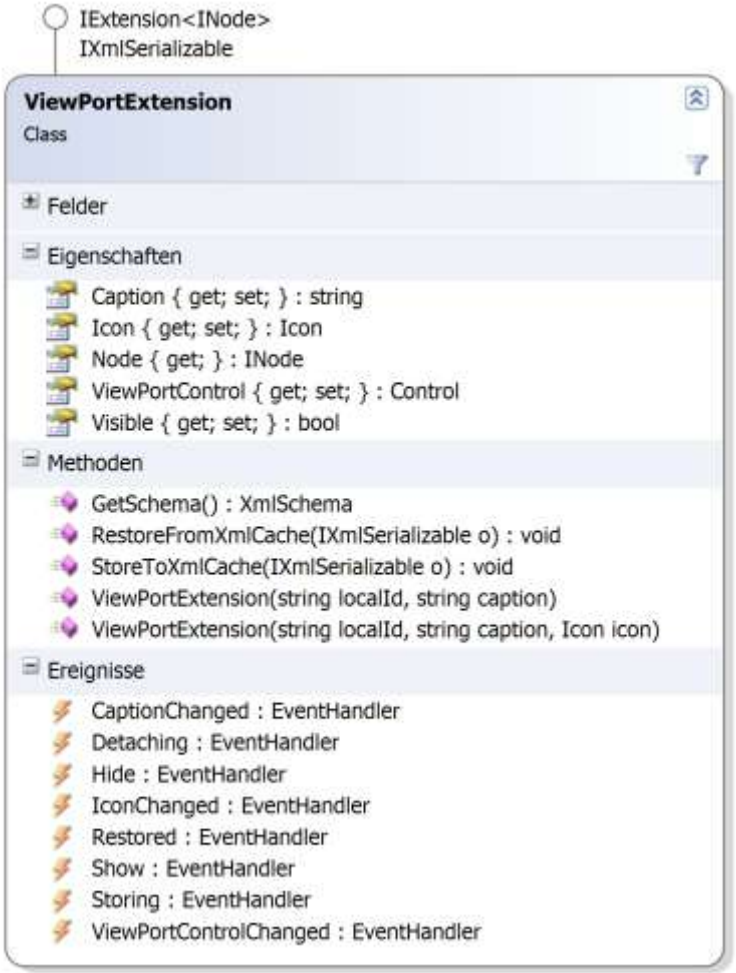


Diagramm 37 DynamicNode.Ext.Visual.ViewPortExtension

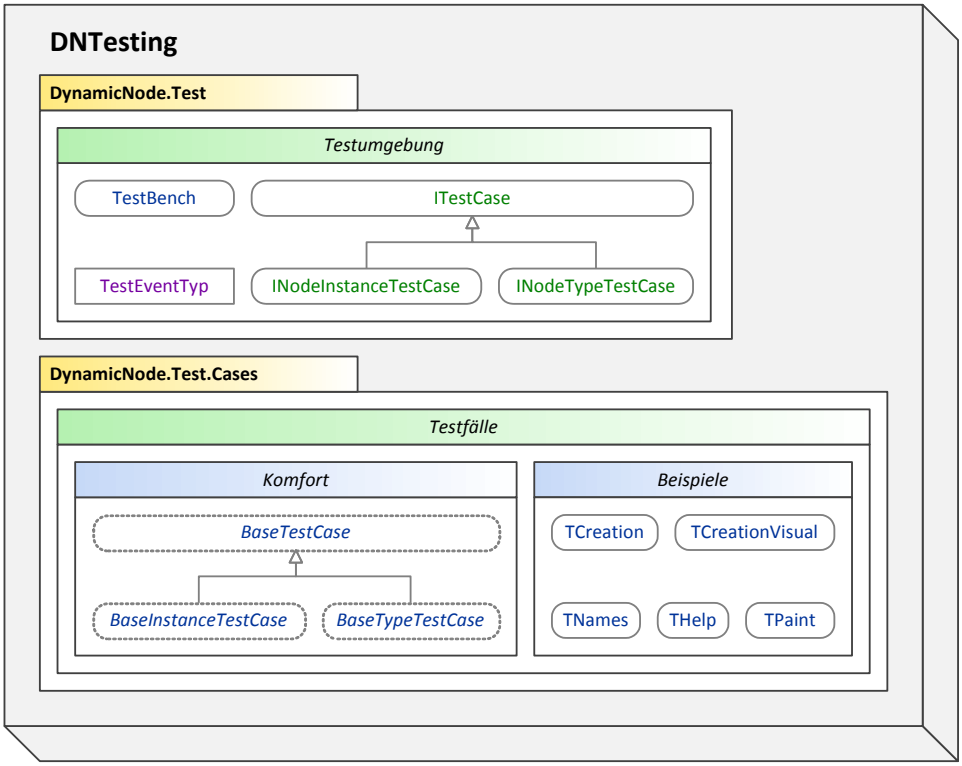


Diagramm 38 Klassenübersicht für DNTesting

Diagramm 39 DynamicNode.Test.ITestCase

Diagramm 40 DynamicNode.Test.INodeInstanceTestCase

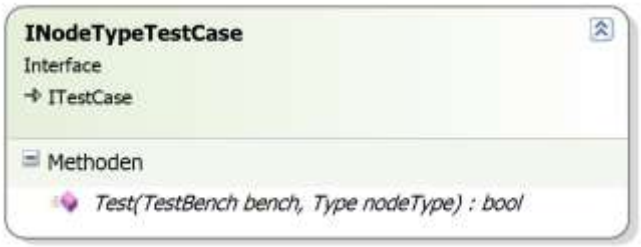


Diagramm 41 DynamicNode.Test.INodeTypeTestCase

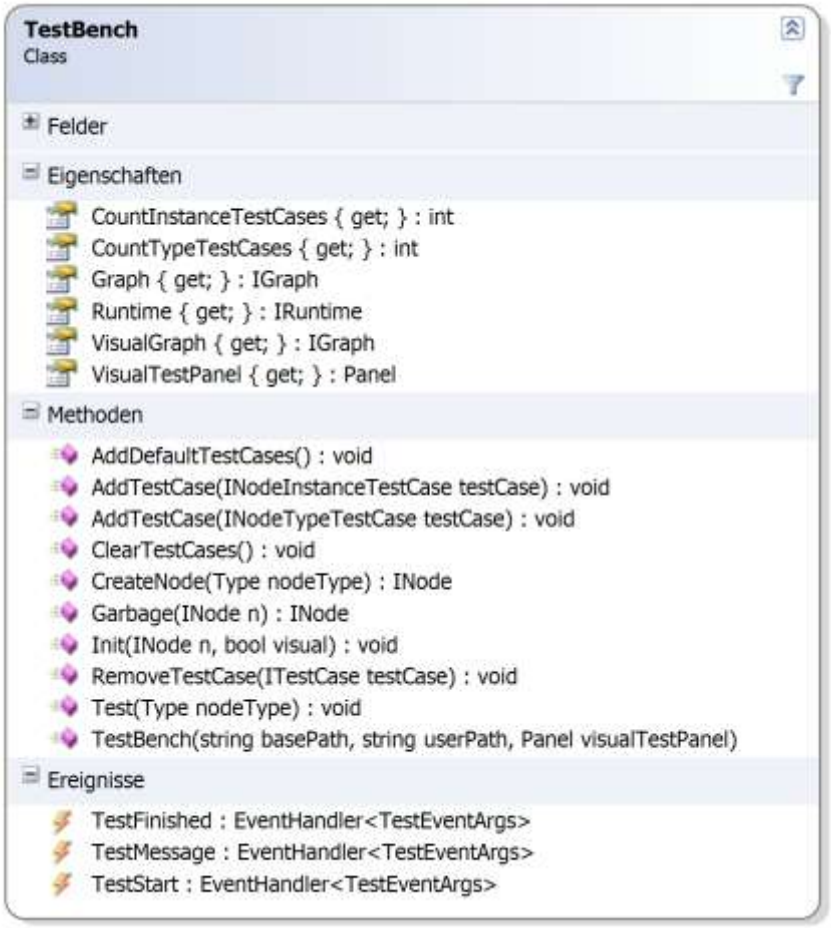


Diagramm 42 DynamicNode.Test.TestBench

Diagramm 43 DynamicNode.Test.TestEventArgs

Diagramm 44 DynamicNode.Test.TestEventTyp

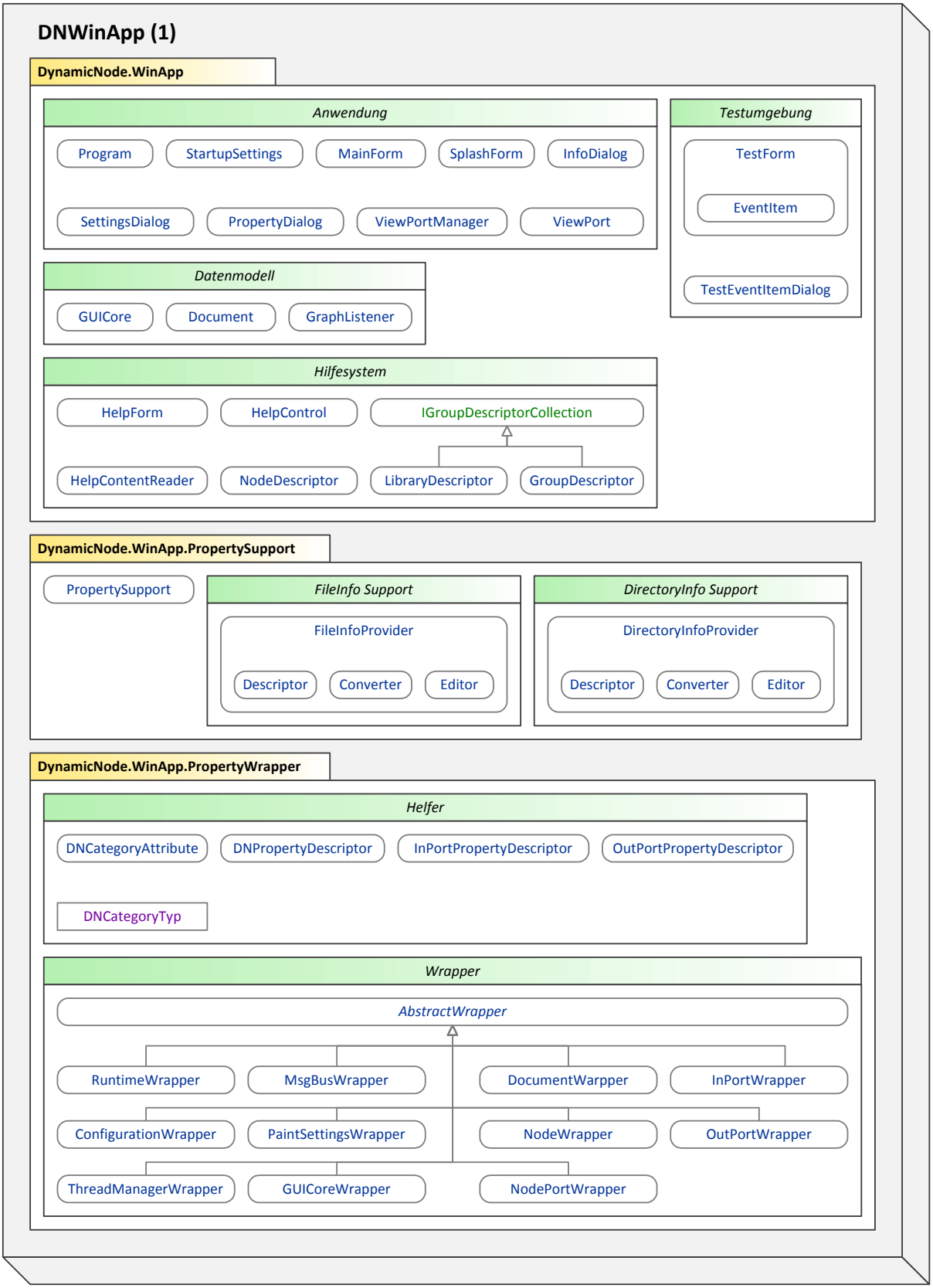


Diagramm 45 Klassenübersicht für DNWinApp Teil 1

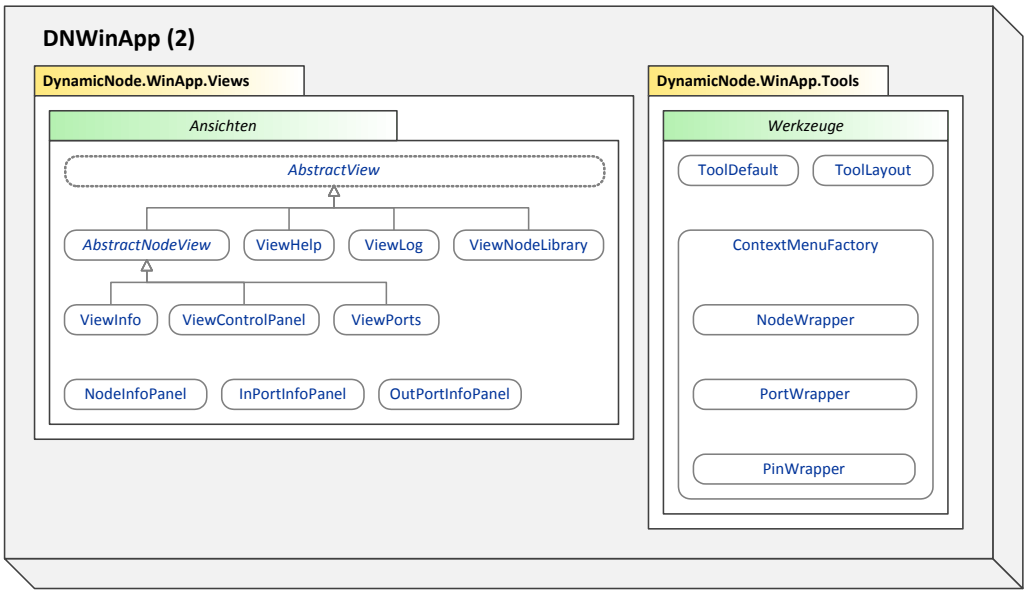


Diagramm 46 Klassenübersicht für DNWinApp Teil 2

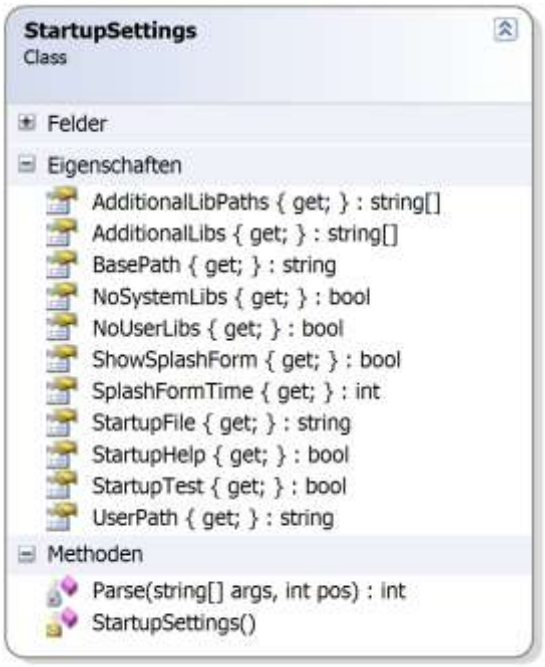


Diagramm 47 DynamicNode.WinApp.StartupSettings

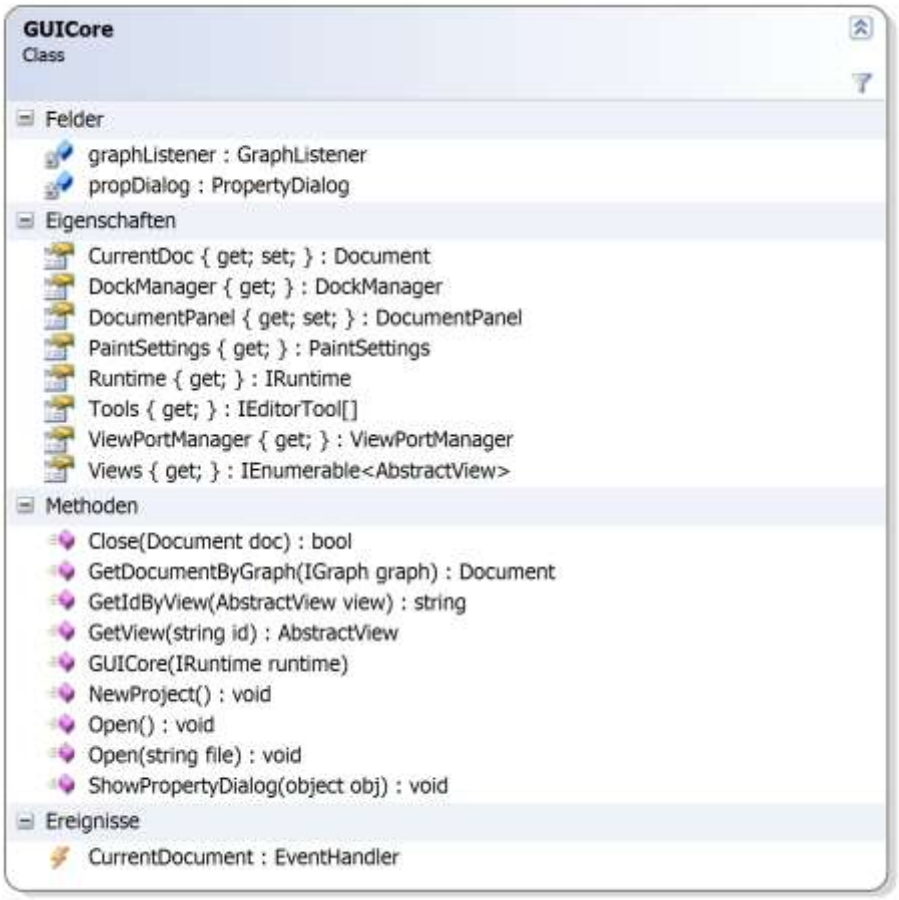


Diagramm 48 DynamicNode.WinApp.GUICore

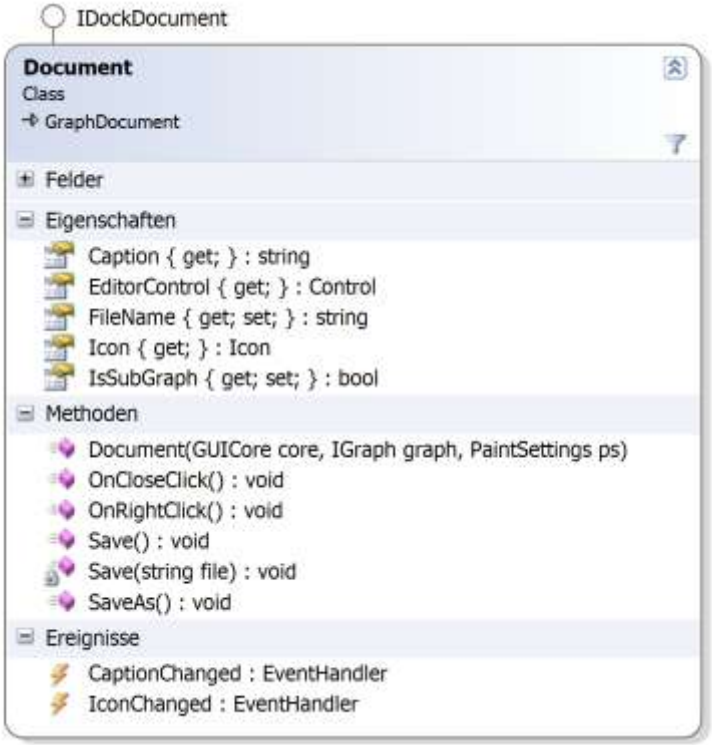


Diagramm 49 DynamicNode.WinApp.Document

Diagramm 50 DynamicNode.WinApp.GraphListener

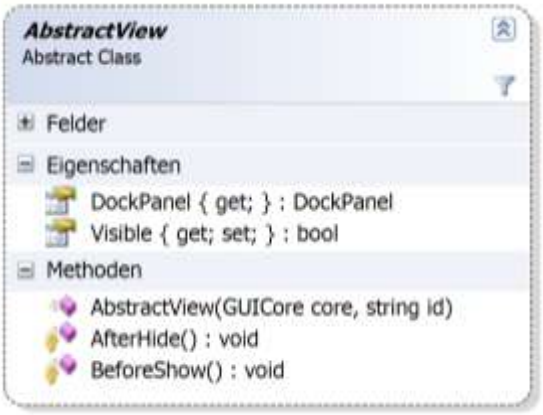


Diagramm 51 DynamicNode.WinApp.AbstractView

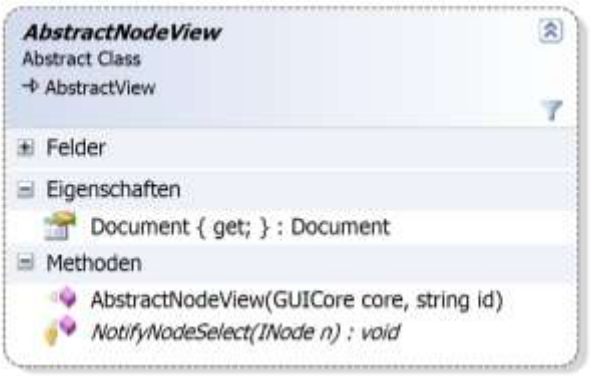


Diagramm 52 DynamicNode.WinApp.AbstractNodeView

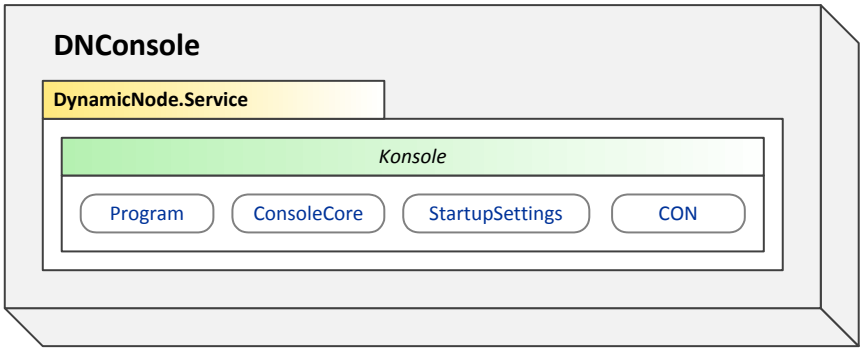


Diagramm 53 Klassenübersicht für DNConsole

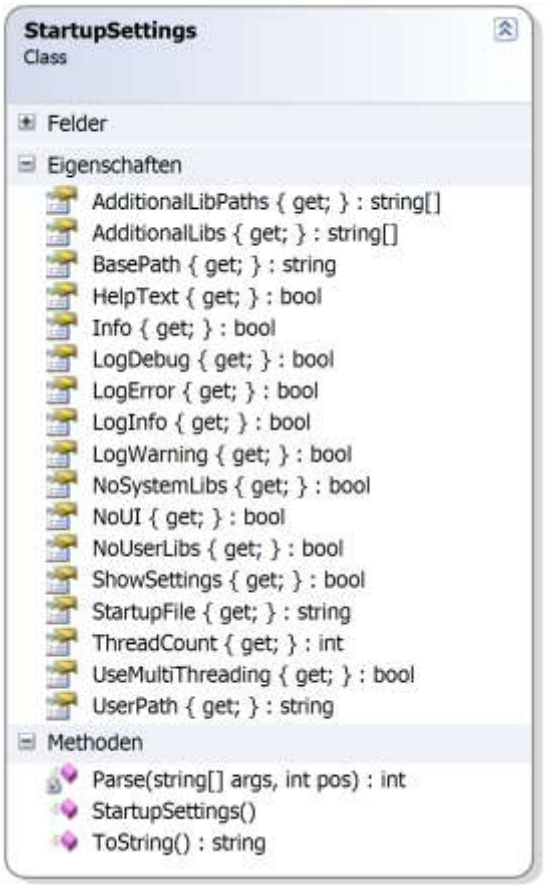


Diagramm 54 DynamicNode.Service.StartupSettings

Diagramm 55 DynamicNode.Service.ConsoleCore

E. Tabellen

Befehlszeilen-Parameter

Parameter	Wirkung
-bp <Pfad> **-basepath <Pfad>**	Setzt den Basis-Pfad für die Laufzeitumgebung. Gewöhnlich zeigt dieser auf das Programmverzeichnis von DynamicNodes.
-up <Pfad> **-userpath <Pfad>**	Setzt den Pfad für das Benutzer-Verzeichnis. Gewöhnlich zeigt dieser auf `%HOMEPATH%\Eigene Dateien\DynamicNodes`
-alp <Pfad> **-addlibpath <Pfad>**	Gibt ein zusätzliches Verzeichnis an, welches nach Knotenbibliotheken durchsucht werden soll.
-lib <Pfad> **-addlib <Pfad>**	Gibt eine zusätzliche Knotenbibliothek zum Laden an.
-nosl **-nosyslibs**	Verhindert das Laden der Knotenbibliotheken im Bibliotheksverzeichnis des DynamicNodes-Basis-Pfades.
-noul **-nouserlibs**	Verhindert das Laden der Knotenbibliotheken im Bibliotheksverzeichnis des DynamicNodes-Benutzer-Pfades.
-h **-help**	Zeigt an Stelle des Haupt-Fensters das Hilfe-Fenster an.
-t **-test**	Zeigt an Stelle des Haupt-Fensters die Testumgebung an.
-ns **-nosplash**	Verhindert das Anzeigen des Begrüßungsfensters beim Start von Dynamic-Nodes.
-st **-splashtime <Zeit>**	Gibt die Zeit in Millisekunden an, für die das Begrüßungsfenster angezeigt werden soll.

Tabelle 1 Befehlszeilen-Parameter für DNWinApp

Parameter	Wirkung
-? **-h** **-help**	Zeigt einen Hilfetext an.
-i **-info**	Zeigt System- und Versionsinformationen an.
-ss **-showsettings**	Zeigt die Startkonfiguration der Laufzeitumgebung an.
-noui	Verhindert interaktive Ausgaben. (sinnvoll z.B. für Batch-Betrieb)
-bp <Pfad> **-basepath <Pfad>**	Setzt den Basis-Pfad für die Laufzeitumgebung. Gewöhnlich zeigt dieser auf das Programmverzeichnis von DynamicNodes.
-up <Pfad> **-userpath <Pfad>**	Setzt den Pfad für das Benutzer-Verzeichnis. Gewöhnlich zeigt dieser auf `%HOMEPATH%\Eigene Dateien\DynamicNodes`
-alp <Pfad> **-addlibpath <Pfad>**	Gibt ein zusätzliches Verzeichnis an, welches nach Knotenbibliotheken durchsucht werden soll.
-lib <Pfad> **-addlib <Pfad>**	Gibt eine zusätzliche Knotenbibliothek zum Laden an.
-nosl **-nosyslibs**	Verhindert das Laden der Knotenbibliotheken im Bibliotheksverzeichnis des DynamicNodes-Basis-Pfades.
-noul **-nouserlibs**	Verhindert das Laden der Knotenbibliotheken im Bibliotheksverzeichnis des DynamicNodes-Benutzer-Pfades.
-noERROR	Verhindert die Ausgabe von Benutzer-Nachrichten der Kategorie ERROR.
-noWARNING	Verhindert die Ausgabe von Benutzer-Nachrichten der Kategorie WARNING.
-noINFO	Verhindert die Ausgabe von Benutzer-Nachrichten der Kategorie INFO.
-debug	Aktiviert die Ausgabe von Benutzer-Nachrichten der Kategorie DEBUG.
-noMT	Schaltet die Parallel-Verarbeitung für die Laufzeitumgebung ab.
-tp <Größe> **-threadpool <Größe>**	Legt die Größe des Thread-Pools fest, mit dem die Laufzeitumgebung die Parallel-Verarbeitung organisiert. Der Standardwert ist die Anzahl der Prozessoren.

Tabelle 2 Befehlszeilen-Parameter für DNConsole

F. Einführung in die Knotenentwicklung

Das Tutorial „Einführung in die Knotenentwicklung“ beginnt auf der nächsten Seite.

G. Quelltexte

Die Quelltexte von DynamicNodes befinden sich in einem zusätzlichen Band.