Einbettung eines Interaktionstypen zur grafischen Bearbeitung von Netzen in VisualWorks® 2.5

Studienarbeit
am
Fachbereich Informatik,
AB Softwaretechnik,
Universität Hamburg

Betreuer:
Prof. Dr. Heinz Züllighoven

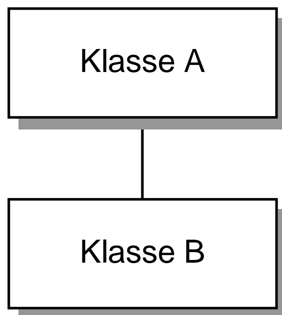
Eike Steffen
Papenstraße 130
22089 Hamburg
Matr.-Nr.: 4 627 462

Sven Lammers
Geschwister-Scholl-Str. 7a
23795 Bad Segeberg
Matr.-Nr.: 4 626 760

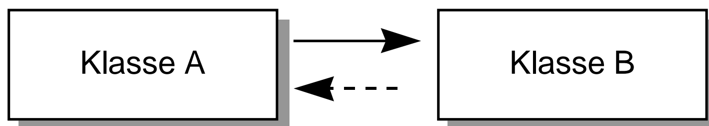
1 Konventionen ... 3
2 Einleitung ... 4
3 Systemgrundlagen ... 6
3.1 Das Model-View-Controller-Paradigma ... 6
3.1.1 Das Model ... 7
3.1.2 Die View ... 8
3.1.3 Der Controller ... 10
3.1.4 Die Interaktion zwischen Model, View und Controller ... 11
3.2 Widgets in VisualWorks® ... 12
3.2.1 Komponenten ... 12
3.2.1.1 Die Klasse VisualComponent ... 13
3.2.1.2 Die Klasse VisualPart ... 13
3.2.1.3 Die Klasse View ... 13
3.2.1.4 Die Klasse Wrapper ... 13
3.2.1.5 Die Klasse CompositePart ... 14
3.2.1.6 Klassenhierarchie ... 15
3.2.2 Die Spezifikationsklassen ... 15
3.3 Der Builder ... 16
3.4 Das Canvas-Tool ... 17
4 Entwurf eines Interaktionstypen zur Bearbeitung von Netzen ... 22
4.1 Anforderungen an den Interaktionstypen ... 22
4.2 Funktionalität des Netz-IAT ... 23
4.3 Implementierung des Interaktionstypen ... 24
4.3.1 Die Funktion der Klasse Net ... 24
4.3.2 Die Klasse NetGrammar ... 25
4.3.3 Die Klasse NetRepresentation ... 26
4.3.4 Die View-Klassen ... 26
4.3.4.1 NetView ... 26
4.3.4.2 NodeView ... 31
4.3.4.3 EdgeView ... 32
4.3.5 Der Net-Controller ... 33
4.3.6 Die NetSpecification ... 37
4.3.7 Beziehungen innerhalb des Interaktionstypen ... 39
4.3.8 Das MVC-Paradigma angewandt auf den Netz-IAT ... 41
4.3.9 Die Anwendungsschnittstelle des Interaktionstypen ... 42
5 Einbettung des Interaktionstypen in VisualWorks® ... 45
5.1 Registrierung des Interaktionstypen in der Palette ... 45
5.2 Funktion der Methode net:into: der Klasse UILookPolicy ... 46
6 Ein Anwendungsbeispiel ... 50
6.1 Ein Netz-IAT zur Bearbeitung von Prozeßmustern ... 50
7 Ausblick ... 56
7.1 Diskussion der Implementierung ... 56
7.2 Mögliche Erweiterungen ... 57
8 Literaturverzeichnis ... 59
9 Anhang (Quellcode) ... 60

1 Konventionen

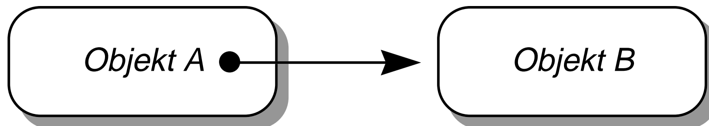
In der hier vorliegenden Arbeit haben wir folgende Konventionen für Fachbegriffe und Zusammenhänge benutzt:



Klasse B *erbt* von Klasse A.



Klasse A *benutzt* Klasse B.
Klasse B *benachrichtigt* KlasseA



Objekt A hält eine Referenz auf Objekt B.

Alle im Text vorkommenden Klassennamen, Variablen, Parameter und Quellcode-Auszügen sind in der Schrift `Courier New` dargestellt.

2 Einleitung

Die meisten gängigen Bibliotheken zur Realisierung von grafischen Benutzungsschnittstellen bieten heute für das Design von Oberflächen vorgefertigte Bausteine an. Zu diesen Bausteinen gehören zum Beispiel Buttons, Dialogfelder und List-Boxen. Diese Art von Bausteinen bezeichnen wir als Interaktionstypen[1], auf die wir im Abschnitt 3 genauer eingehen werden. Mit Hilfe dieser Interaktionstypen kann man zum Beispiel identische Informationen auf verschiedene Weisen darstellen. Dieses bedeutet, daß ein Interaktionstyp alleine für die eigentliche Art der Darstellung verantwortlich ist.

In der hier vorliegenden Arbeit haben wir uns mit dem Entwurf und der Implementierung eines Interaktionstypen befaßt, der in der Lage ist, ein Netz darzustellen, und Möglichkeiten zur Manipulation des Netzes bietet. Für die Implementierung dieses Interaktionstypen haben wir die Smalltalk Programmierumgebung VisualWorks®-System 2.5 verwendet. Da es sich hierbei um eine Erweiterung des VisualWorks®-System handelt, liegt dieser Arbeit das Model-View-Controller Paradigma zugrunde. Im Hinblick auf Gemeinsamkeiten zwischen dem MVC-Paradigma und der am Arbeitsbereich Softwaretechnik gelehrten Metapher Werkzeug - Automat - Material (WAM) möchten wir auf [8] verweisen.

Um die Art der Erweiterung des Smalltalk-Systems deutlich zu machen, werden wir zunächst im Kapitel 2 die technischen Grundlagen vorstellen. In diesem Abschnitt werden wir das MVC-Paradigma behandeln und erläutern, auf welche Weise Widgets[2] in dem VisualWorks®-System aufgebaut sind. Desweiteren werden wir die Aufgaben und Struktur des sogenannten Builder vorstellen. Abschließend beschreiben wir in diesem Kapitel die Struktur des Canvas-Tools, mit dem die Oberflächen und Dialoge der Anwendungen gestaltet werden können. Im Anschluß hieran werden wir im Kapitel 3 die Implementierung unseres Netz-IAT's beschreiben. Innerhalb dieses Abschnittes erläutern wir zunächst die Anforderungen, die von uns an einen IAT gestellt werden, so wie die Funktionalität, die wir erreichen wollen. Den Abschluß des dritten Kapitels bildet dann die Erläuterung der eigentlichen Implementierung, inklusive der Schnittstellenbeschreibung und der Klassenhierarchie. Die Art und Weise, wie wir den IAT dann in das VisualWorks®-System eingebettet haben, greifen wir dann im Kapitel 4 auf.

Kapitel 5 beschreibt auf der Basis des in den ersten Kapiteln entwickelten Interaktionstypen eine Vorgehensweise, mit der wir aus dieser allgemeinen Grundlage einen speziellen Netz-IAT ableiten. Grundlage hierfür ist ein Ausschnitt der Promotion von Guido Gryzcan, in dem die Vergegenständlichung kooperativer Arbeit durch Prozeßmuster behandelt wird ([6] Kapitel 7). Hierfür fassen wir ein Prozeßmuster als die Spezialisierung eines allgemeinen Netzes auf, so daß der hier vorgestellte IAT in der Lage ist, Prozeßmuster zu editieren und zu verändern.

---

[1] Im weiteren Verlauf der Arbeit werden wir alternativ zu der Bezeichnung des Interaktionstypen die Kurzform IAT verwenden.

[2] Unter Widgets verstehen wir grafische Komponenten, die zur Erstellung von Oberflächen verwendet werden können.

Den Abschluß dieser Arbeit bildet die Diskussion der Implementierung unseres Interaktionstypen, insbesondere hinsichtlich von Problemen des von uns gewählten Entwurfes. Desweiteren werden wir in diesem Abschnitt die Beschränkungen durch die dem Smalltalk-System zugrunde liegenden Mechanismen beschreiben. Der letzte in diesem Abschnitt beschriebene Aspekt ist der der möglichen Erweiterung des Netz-Interaktionstypen hinsichtlich der Funktionalität und der von uns verwendeten Konzepte.

An dieser Stelle möchten wir besonders Prof. Dr. Heinz Züllighoven und Dr. Guido Gryczan für die konstruktive Unterstützung bei der Erstellung dieser Arbeit danken. Insbesondere gilt unser Dank auch Carola Lilienthal, die zum Gelingen unserer Arbeit durch lebhafte Diskussionen beigetragen hat.

3 Systemgrundlagen

In diesem Abschnitt werden von uns die grundlegenden Prinzipien der Softwareentwicklung in dem VisualWorks®-System beschrieben und erläutert. Grundlage dieses Kapitels sind im wesentlichen [2], [1] sowie die Object Reference [3] und das Cookbook [4] des VisualWorks®-Systems 2.5.

3.1 Das Model-View-Controller-Paradigma

Der Entwicklung von Software mit dem VisualWorks®-System liegt das MVC-Paradigma zugrunde, bei der eine Anwendung in drei Teile unterteilt wird - Model, View und Controller. Mit Hilfe dieses Paradigmas sind wir in der Lage, Anwendungen so zu strukturieren, daß man die Bestandteile der Software nach Aufgaben unterscheiden kann. Es sei von vorne herein bemerkt, daß bei der Verwendung der Begriffe Model, View und Controller nicht ausschließlich Exemplare der in dem VisualWorks®-System vorhandenen Klassen mit den Namen Model, View und Controller gemeint sind. Vielmehr werden durch Model, View und Controller Aufgaben von Objekten in einer Smalltalk Anwendung beschrieben.

Bevor wir im einzelnen auf die Komponenten eingehen, geben wir im folgenden eine kurze Erläuterung von Model, View und Controller:

- Das Model ist ein Objekt, in dem die auf der Oberfläche darzustellende Information hinterlegt ist. Diese Information wird einer View auf Anforderung zur Verfügung gestellt. Desweiteren muß das Model auf Eingaben des Benutzers reagieren, die von der Oberfläche an die Anwendung weitergereicht werden.

- Innerhalb der View wird festgelegt, wie die Information des Models auf einem Ausgabemedium, einem sogenannten `GraphicsContext` dargestellt wird. Dieser `GraphicsContext` kann zum Beispiel ein Window oder ein Drucker sein.

- Der Controller ist ein der View direkt zugeordnetes Objekt, welches die Interaktion mit dem Benutzer steuert und die aufgetretenen Events in Nachrichten umwandelt, die entweder der View oder dem Model geschickt werden. Die von dem Controller zu verarbeitenden Events bestehen im wesentlichen aus Mausaktionen, Tastatureingaben oder Aktionen aus einem Menü heraus.

Im Hinblick auf den Entwurf von Smalltalk Anwendungen weisen wir darauf hin, daß es innerhalb dieser Anwendung meistens mehr als ein Model gibt. Das gleiche gilt auch für Objekte, die die Rolle eines Controllers oder einer View übernehmen.

3.1.1 Das Model

Im Bezug auf die in Abschnitt 2.1 angeführte Kurzbeschreibung der Aufgabe eines Models im Model-View-Controller-Kontext ist bis jetzt die Frage noch ungeklärt, welchen Typ Objekte haben, die als Model fungieren. Grundsätzlich ist hierzu zu sagen, daß jedes Objekt einer beliebigen Klasse die Funktion des Models übernehmen kann, wenn in ihr folgendes Verhalten definiert ist:

- Innerhalb des Models wird die domänenspezifische Information gespeichert und verändert. Unter der domänenspezifischen Information verstehen wir diejenigen Daten, die unmittelbar zu dem Anwendungsbereich gehören. Bei dem Model unseres Netz-IAT's zum Beispiel werden hier die Knoten und Kanten des Netzes vorgehalten. Somit muß in der entsprechenden Klasse eine Schnittstelle definiert sein, mit der auf die domänenspezifische Information zugegriffen werden kann und die es erlaubt, Werte zu ändern. Weder in View-Objekten noch in Controller-Objekten werden diese Daten abgespeichert oder manipuliert.

- Beliebige andere Objekte müssen die Option haben, sich als sogenannter Dependent in einem Model registrieren zu lassen. Änderungen am Model werden den Dependents dann über einen speziellen Benachrichtigungsmechanismus mitgeteilt.

Die vorgenannten Eigenschaften werden in den Klassen `Object` und `Model` definiert. Der Unterschied zwischen den beiden Klassen besteht vornehmlich darin, daß in der Klasse `Model` eine bessere Unterstützung des Benachrichtigungsmechanismus implementiert ist. Das Ziel bei der Einführung des Benachrichtigungsmechanismus innerhalb des Model-View-Controller-Paradigmas ist, die Unabhängigkeit zwischen den Model-Objekten und allen anderen Objekten, die auf den im Model enthaltenen Informationen aufbauen, gewährleisten zu können. Im einzelnen bedeutet dies, daß das Model keine näheren Einzelheiten der an dem Model interessierten Objekte kennen soll. Die Dependents auf der anderen Seite sind darauf angewiesen, über eventuelle Änderungen am Model sofort informiert zu werden. Diese Benachrichtigung der Dependents erfolgt über die im Model gehaltene Dependency-Liste mit Hilfe der change-update-Methoden.

Die zentralen Methoden sind hierbei:

- `changed`
- `changed: aChangedSymbol`
- `update: aChangedSymbol`
- `update: aChangedSymbol with: anotherObject`

Die `changed`-Methoden werden vom jeweiligen Model benutzt, um an den in der Dependency-Liste eingetragenen Objekten die entsprechende `update`-Methode aufzurufen. Der Parameter `aChangedSymbol` repräsentiert die Art der Veränderung am Model in Form eines Symbols (zum Beispiel: `#insertNewNode`, `#changedPreferredBounds`, etc.).

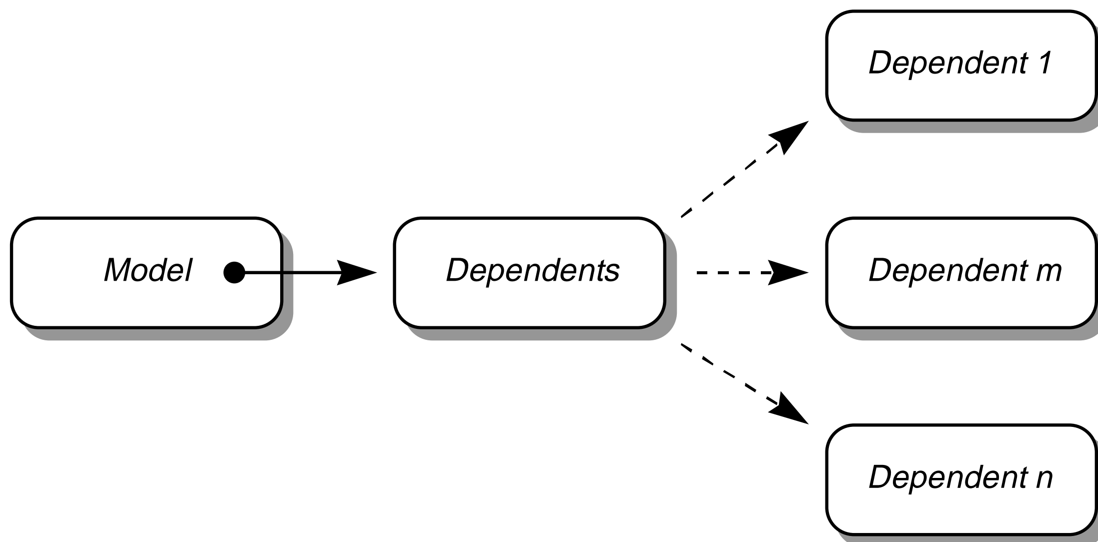


Abbildung 2.1

Abbildung 2.1 zeigt das Verhalten eines Models, nachdem ein Ereignis eingetreten ist, das die Weitergabe der hinterlegten Informationen an die einzelnen Dependents notwendig macht. Zuerst greift das Model über eine Objektreferenz auf die Liste der abhängigen Objekte zu. Danach wird an jedem Element der Dependency-Liste die entsprechende `update`-Methode aufgerufen. Sowie ein Dependent über die Änderung des Models informiert wurde, obliegt es dem Dependent selber, ob er sich die veränderten Informationen über einen Methodenaufruf an dem Model holt oder nicht.

3.1.2 Die View

Die View stellt eine bestimmte Sichtweise auf ein Objekt dar, welches die Rolle eines Models übernommen hat. Somit ist es durchaus möglich und üblich, daß für ein Model mehrere Views existieren, die jeweils eine andere Sicht auf die im Model hinterlegte Information darstellen. Die Aufgabe einer View besteht also darin, den Zustand des zugrunde liegenden Models zu reflektieren. Als Dependent muß die View also insbesondere auf die `update`-Methoden reagieren, die von Model-Objekten aufgerufen werden, bei denen sie eingetragen ist.

Im VisualWorks®-System erben alle speziellen Views von der abstrakten Klasse `View`. Diese wiederum ist die Subklasse von `DependentPart`. In dieser ebenfalls abstrakten Klasse `DependentPart` ist dasjenige Protokoll implementiert, welches notwendig ist, um die Abhängigkeit zwischen einer View und einem Model herzustellen. Die Klasse `View` selbst fügt die Schnittstelle hinzu, die es ermöglicht, Informationen auf einem `GraphicsContext` darzustellen. Aufgrund dieser Konstruktion innerhalb des VisualWorks®-Systems ist die Funktionalität einer View auf zwei abstrakte Klassen verteilt. Die Aufgaben einer View können also nur von solchen Objekten übernommen werden, die ein Exemplar der Klasse `View` selbst oder einer ihrer Unterklassen sind.

Um die vorgenannten Aufgaben erfüllen zu können, muß die Schnittstelle einer View folgende Methoden aufweisen:

- `invalidate`
  Diese Methode sorgt dafür, daß sich die View innerhalb der `preferredBounds` neu zeichnet. Der Aufruf erfolgt normalerweise immer zu dem Zeitpunkt, an dem sich eine Eigenschaft der View oder des Models geändert hat.

- `displayOn:`
  Innerhalb dieser Routine wird festgelegt, welche Informationen wie und wo auf einem `GraphicsContext` dargestellt wird. Der aktuelle `GraphicsContext` wird dieser Methode als Parameter beim Aufruf mitgegeben werden.

- `preferredBounds`
  Durch den Aufruf dieser Methode kann man die von einer View im Normalfall verwendete Fläche ermitteln, die ausreicht, um die gesamte Information darstellen zu können. Die View liefert hier ein `Rectangle` zurück, welches die linke obere Ecke und die rechte untere Ecke der Grafik umfaßt. Analog zu der Methode `bounds` handelt es sich auch hier um relative Koordinatenangaben.

- `bounds`
  Der Aufruf dieser Methode liefert ein `Rectangle` zurück, daß die tatsächlichen Ausmaße der darzustellenden grafischen Informationen beinhaltet. Es handelt sich hierbei um relative Koordinatenangaben, so daß man vor der Verwendung dieser Daten eine Umrechnung auf die absoluten Koordinaten vornehmen muß.

- `container`
  Die Anwendung dieser Methode erlaubt es, Informationen über die Struktur abzurufen, in die die View eingebettet ist. Wir werden auf diese Methode in Abschnitt 2.2.1 genauer eingehen.

- `component`
  Die Anwendung dieser Methode erlaubt es, Informationen über die Struktur abzurufen, in die die View eingebettet ist. Wir werden auf diese Methode in Abschnitt 2.2.1 genauer eingehen.

- `defaultControllerClass`
  Die Bedeutung dieser Methode liegt in der Tatsache begründet, daß während der Erzeugung eines View-Objektes ein Exemplar des dazugehörigen Controllers erzeugt und dem View-Objekt zugewiesen wird. Hierfür ist in dieser Methode der Name der im Regelfall einzusetzenden Controller-Klasse hinterlegt.

3.1.3 Der Controller

Wie im Abschnitt über die View erläutert, kann einer View ein Controller zugeordnet werden. Dieser Controller übernimmt in bestimmten Situationen die Kontrolle bezüglich der Interaktion mit einem Benutzer. Liegt der Focus der Anwendung auf einer View, so kann der der View zugeordnete Controller die Kontrolle über die Eingaben übernehmen. Verliert die View den Focus, so gibt der Controller die Kontrolle wieder ab. In Abhängigkeit von der Art der Eingaben entscheidet der Controller, ob dieses Event dem zugrunde liegenden Model oder gar der entsprechenden View direkt zugesendet werden muß.

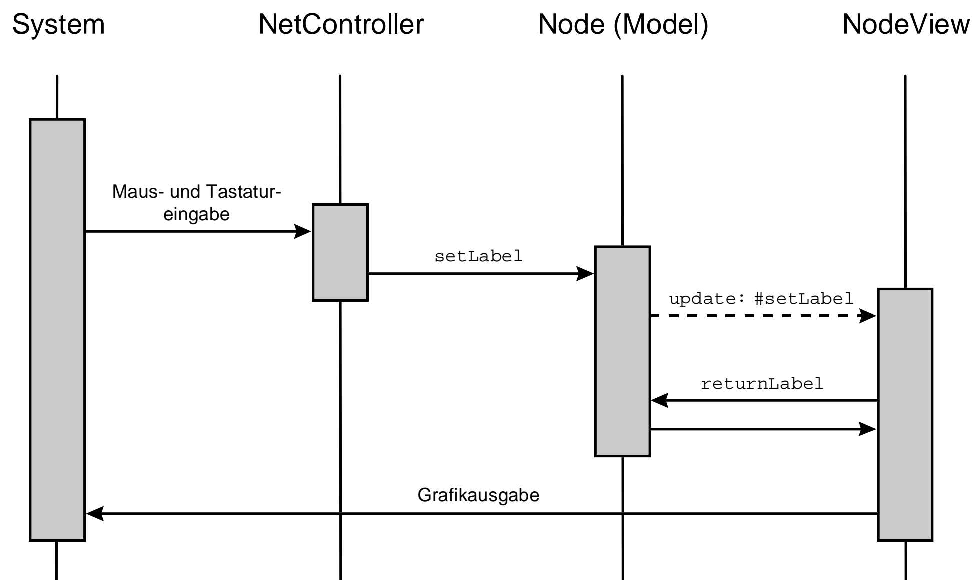


Abbildung 2.2

In Anlehnung an [1] möchten wir beispielhaft die Verarbeitung eines Events durch einen Controller beschreiben:

1. Der Controller erkennt, daß eine Eingabe vorliegt.
2. Der Controller schickt daraufhin eine Nachricht an das Model.
3. Das Model führt die zugehörige Methode aus und benachrichtigt seine Dependents, daß sich sein Zustand geändert hat. Wenn eine Bildschirmdarstellung dynamisch an Änderungen des Models angepaßt werden soll, ist die View normalerweise ein Dependent des Models und erhält so eine `#update`-Nachricht.
4. Die View fragt beim Model die aktuellen Informationen für die Grafikausgabe ab und erzeugt eine neue Grafik.
5. Die so neu erzeugte Grafik wird dann auf dem Bildschirm ausgegeben.

Die Wege des Kontrollflusses werden in der Abbildung 2.2 verdeutlicht.

Objekte, die die Rolle eines Controllers übernehmen sollen, sind grundsätzlich Exemplare der Klassen `Controller` oder `ControllerWithMenu`. In diesen Klassen sind die Methoden definiert, die die Verarbeitung externer Ereignisse ermöglichen. Betrachtet man die Arbeitsweise eines Controllers genauer, so erkennt man, daß sich ein aktiver Controller in einer sogenannten `controlLoop` befindet.

Diese Schleife gliedert sich grundsätzlich in die folgenden drei Teile:

1. Der erste Schritt besteht darin, daß kurzzeitig die Kontrolle an die übergeordnete Controller-Ebene, den `WindowController`, abgegeben wird, um sicherzustellen, daß die Window-Events ausgeführt werden können. Diese Ereignisse bestehen zum Beispiel in dem Neuzeichnen oder Schließen eines Fensters.
2. Hiernach wird überprüft, ob der Controller die Kontrolle abgeben soll oder ob die Kontrolle bei dem zur Zeit aktiven Controller verbleibt.
3. Im letzten Schritt wird der `EventChannel` auf Benutzereingaben überprüft. Liegen Benutzereingaben vor, so werden diese in Abhängigkeit von der Art der Ereignisse weiterverarbeitet.

Den ersten Teil dieser ständig durchlaufenden Schleife erledigt ein Controller durch den Aufruf der Methode `poll`. Hierdurch wird, wie bereits geschildert, die Kontrolle kurzzeitig abgegeben, um Window-Events abarbeiten zu können. Daraufhin folgt der Aufruf der Methode `isControlActive`, mit deren Hilfe der Controller entscheidet, wann die Kontrolle abzugeben oder zu behalten ist. Aus diesem Grund ist es notwendig, daß jeder Controller diese Methode implementiert. Die meisten Controller versuchen, die Kontrolle solange zu behalten, bis die dazugehörige View den Focus der Anwendung verliert. Im letzten Schritt dieser Schleife schließlich arbeitet der Controller über die Methode `controlActivity` die anstehenden Ereignisse ab. Da verschiedene Controller auf Benutzereingaben anders reagieren müssen, ist diese Methode individuell für jede Controller-Klasse zu implementieren. Nachdem die Ausführung von `controlActivity` beendet ist, wird diese dreistufige Schleife erneut abgearbeitet.

3.1.4 Die Interaktion zwischen Model, View und Controller

Nachdem wir uns in den vorangegangenen Abschnitten jeweils allein auf das Model, die View und den Controller konzentriert haben, möchten wir abschließend die Kommunikation zwischen diesen drei Bestandteilen aufgreifen. Die nachfolgende Abbildung 2.3 zeigt die enge Verbindung von Objekten, die die verschiedenen Rollen innerhalb des MVC-Paradigmas darstellen. Diese Verbindungen untereinander sind notwendig, da Model, View und Controller ständig Nachrichten austauschen müssen, um adäquat auf neue Situationen bzw. Zustände reagieren zu können.

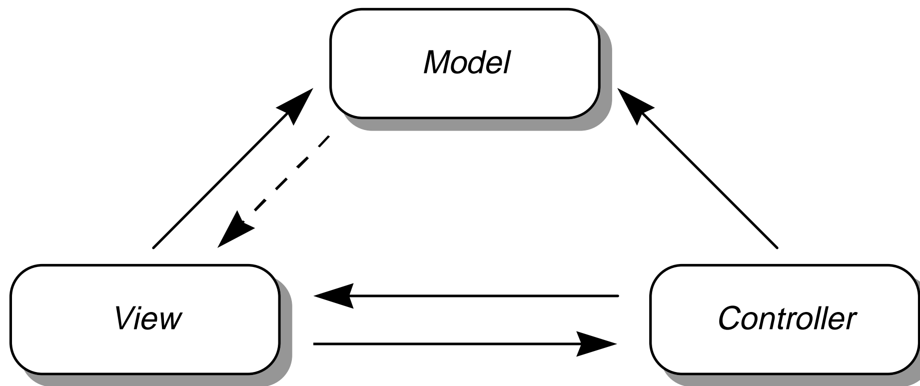


Abbildung 2.3

Die gerichteten Kanten zwischen den einzelnen Objekten bedeuten in diesem Fall die Möglichkeit, einem anderen Objekt eine Nachricht zukommen zu lassen. Für die Implementierung des MVC-Paradigmas bedeutet dies, daß eine View sowohl eine Referenz auf ihren Controller als auch auf das ihr zugeordnete Model besitzen muß. Analog hierzu besitzt ein Controller Referenzen auf die View und das Model. Lediglich das Model kann nur die View referenzieren. Erhält zum Beispiel der Controller ein Event, so muß er entscheiden, von wem dieses abgearbeitet werden soll. Zu diesem Zweck hat er die Möglichkeit, dieses direkt an das Model zu senden und / oder die View für die Abarbeitung des Ereignisses heranzuziehen. In gleicher Art und Weise entscheidet die View, welches Objekt - Controller und / oder Model - für die Erledigung einer Aufgabe benachrichtigt werden muß. Eine Ausnahme hierbei bildet das Model, welches ausschließlich der View Nachrichten senden kann.

3.2 Widgets in VisualWorks®

In diesem Kapitel wollen wir erläutern, welche Klassen für den Aufbau von grafischen Benutzeroberflächen innerhalb des VisualWorks®-System verwendet werden und in welcher Beziehung sie zueinander stehen.

3.2.1 Komponenten

Zunächst werden wir die wichtigsten Klassen vorstellen, aus denen sich die grafischen Elemente zur Gestaltung einer Oberfläche in VisualWorks® zusammensetzen. Im einzelnen sind dies die Klassen: `VisualComponent`, `VisualPart`, `View` und `Wrapper`. Dabei verstehen wir in VisualWorks® unter einem grafischen Element ein Objekt, welches die Möglichkeit besitzt, sich selbst auf einem beliebigen `GraphicsContext` darzustellen.

3.2.1.1 Die Klasse VisualComponent

Alle grafischen Elemente einer Oberfläche erben von der abstrakten Klasse `VisualComponent`. Hierdurch wird den Elementen der Oberfläche die Basisfunktionalität in Form des *visual component protocol* zur Verfügung gestellt. Dieses Protokoll besteht aus einer Reihe von Nachrichten, die den grafischen Aufbau und die Kommunikation zwischen den Komponenten regeln. Die wichtigsten beiden Nachrichten sind `displayOn:` und `preferredBounds`. Diese beiden Nachrichten müssen von jedem grafischen Objekt verstanden werden. Nähere Erläuterungen zu diesen beiden Routinen stehen im Kapitel 3.

3.2.1.2 Die Klasse VisualPart

Die abstrakte Klasse `VisualPart`, eine Unterklasse von `VisualComponent`, implementiert das Verhalten und den Zustand eines auf einem `GraphicsContext` sichtbaren Elementes.

Ein Objekt vom Typ `VisualPart` baut eine Beziehung auf zwischen einem sogenannten Container und der darin enthaltenen Komponente. Dieses bedeutet, daß ein `VisualPart`-Objekt zum einen eine Referenz auf eine in ihm enthaltene Komponente besitzt und zum anderen eine Referenz auf einen Container enthält. Die in der Exemplarvariable `container` enthaltene Referenz zeigt auf ein Objekt, welches das `VisualPart`-Objekt einhüllt. In der Regel ist dieser Behälter ein Wrapper. Mit Hilfe von `VisualParts` und Wrappern ist es möglich, baumartige Strukturen von grafischen Elementen aufzubauen und zu verwalten.

3.2.1.3 Die Klasse View

Die abstrakte Klasse `DependentPart` ist eine Unterklasse von `VisualPart` und ergänzt diese um die Fähigkeit, mit Objekten, die als Model gelten, umzugehen und zu kommunizieren. `DependentPart` hat nur die eine Unterklasse `View`. Diese abstrakte Klasse besitzt zusätzlich die Fähigkeit, mit Controllern umzugehen.

3.2.1.4 Die Klasse Wrapper

Wrapper sind Container, die jeweils nur eine Komponente enthalten. Sie haben die Aufgabe, bestimmte Dienste für die enthaltene Komponente zu erbringen. Diese Dienste können zum Beispiel die Transformation von Koordinaten, das Zeichnen von Rahmen oder Scrollbalken sein. Alle Wrapper sind Unterklassen von der Klasse `Wrapper`, die ihrerseits eine Unterklasse von `VisualPart` ist. Wie alle Objekte vom Typ `VisualPart`, kann sich ein Wrapper selbst in einem Fenster zeichnen. Aber nicht alle Wrapper müssen eine grafische Repräsentation besitzen. Ein `TranslatingWrapper` zum Beispiel verändert nur die übergebenen Koordinaten, hat aber selbst keine Form der grafischen Darstellung.

Wrapper werden in VisualWorks sehr häufig benutzt, denn mit ihnen lassen sich baumartige Strukturen von grafischen Komponenten aufbauen und verwalten. Dieses ist möglich, da ein Wrapper auch einen Wrapper als Komponente enthalten kann. Abbildung 2.4 zeigt exemplarisch eine solche Struktur.

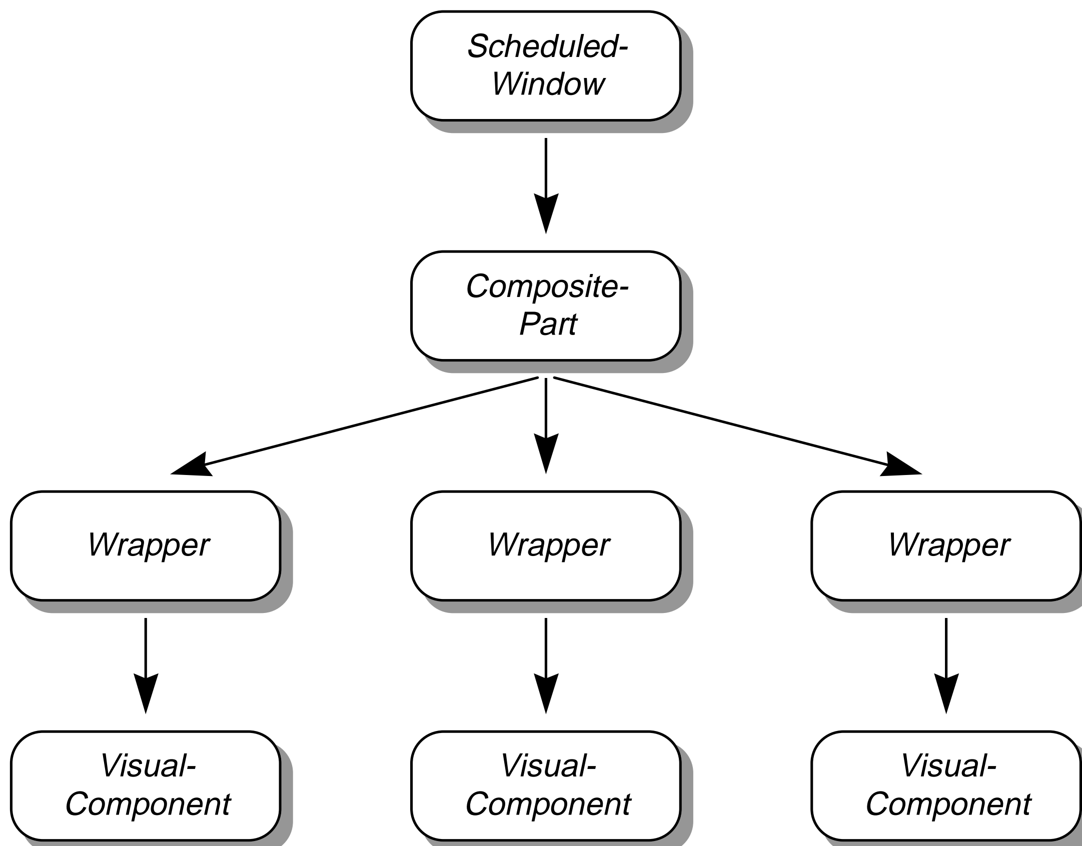


Abbildung 2.4

3.2.1.5 Die Klasse CompositePart

Zum Aufbau von komplizierten Grafiken, die aus vielen einzelnen Bestandteilen bestehen, ist es sinnvoll, diese aus mehreren einzelnen Grafikobjekten zusammenzusetzen. Für den Aufbau von hierarchisch aufgebauten Grafiken ist die Klasse `CompositePart` geeignet. Ein Objekt vom Typ `CompositePart` besitzt eine Exemplarvariable `components`, in der es eine Collection von Wrappern speichert. `CompositePart` ist eine Unterklasse von `VisualPart`. Alle Methodenaufrufe werden von einem `CompositePart` an die in ihm enthaltenen Wrapper weitergeleitet. Weiterhin koordiniert ein `CompositePart` die weitere Verarbeitung der Rückgabewerte der Methodenaufrufe.

3.2.1.6 Klassenhierarchie

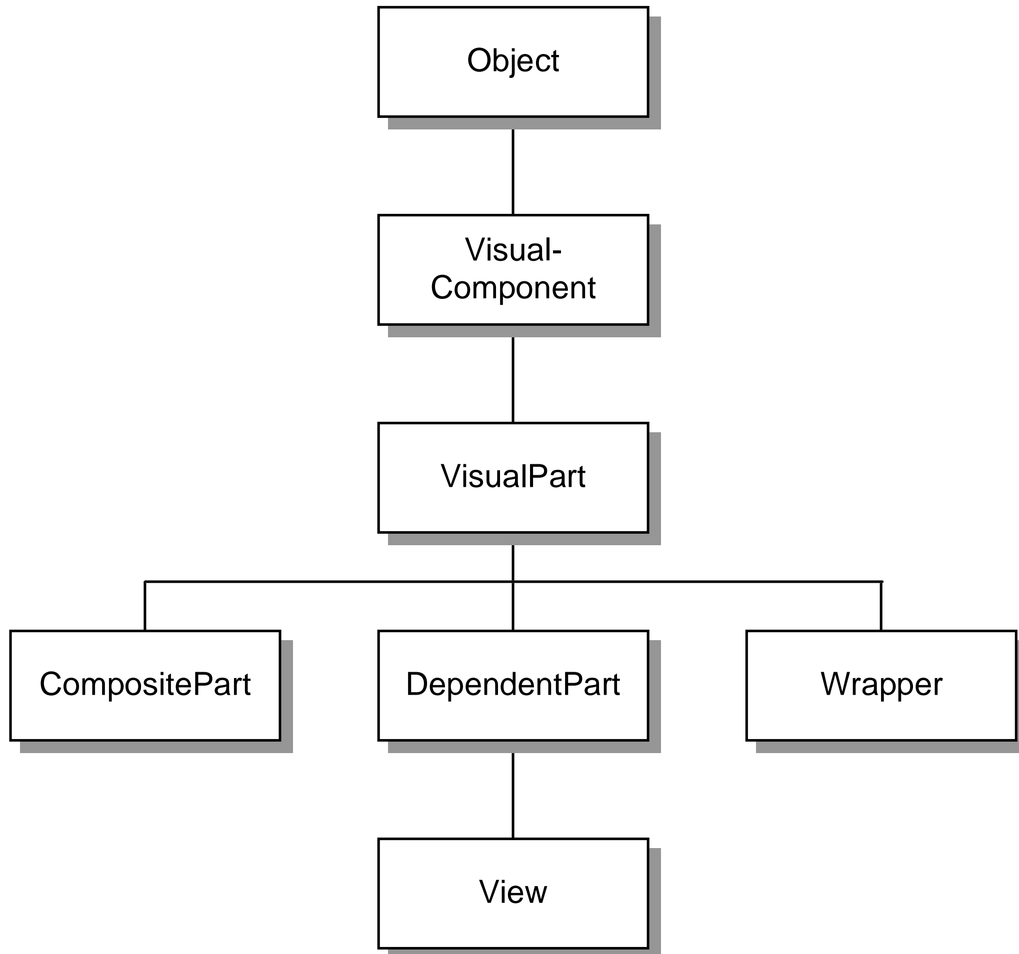


Abbildung 2.5

Die Beziehung der in den vorherigen Abschnitten beschriebenen Klassen möchten wir in der Abbildung 2.5 in Form der Klassenhierarchie aufzeigen.

3.2.2 Die Spezifikationsklassen

Die Spezifikationsklassen in VisualWorks® dienen der Beschreibung von Eigenschaften und Funktionen der grafischen Komponenten wie zum Beispiel Buttons oder Edit-Felder. Jedes dieser für die Oberflächengestaltung verfügbaren Elemente hat ihre eigene Spezifikationsklasse. `ComponentSpec` ist die abstrakte Oberklasse aller Spezifikationsklassen, die grafische Komponenten beschreiben. Sie enthält zum Beispiel Informationen über das Layout der Komponente, welche unter anderem die Position der Komponente in einem Fenster beschreibt. Die Unterklassen `NamedSpec` und `WidgetSpec` beschreiben die Eigenschaften von passiven bzw. aktiven Komponenten. Zu diesen Eigenschaften gehören Beispiel Name, Farbe, Rahmendekoration und das zu verwendende Model.

Mit Hilfe der Beschreibung der Komponenten durch Spezifikationsklassen können zur Laufzeit eines Programmes konkrete Exemplare der grafischen Komponenten erzeugt werden. VisualWorks® speichert zum Beispiel die mit Hilfe des Canvas-Tools erstellten Oberflächen nicht direkt als ein grafisches Objekt, sondern es serialisiert die Bestandteile der Oberfläche, in dem nur die Namen der Spezifikationsklassen mit den dazugehörigen Parametern (Position, Farbe etc.). Zur Laufzeit des Programmes reichen diese Informationen aus, um die Objekte der Oberfläche zu erzeugen.

3.3 Der Builder

In diesem Abschnitt soll die Rolle der Builder in VisualWorks® näher erläutert werden.

Ein Builder ist ein Exemplar der Klasse `UIBuilder` und erfüllt drei Aufgaben im System:

- Erzeugung einer Benutzeroberfläche gemäß vorgegebener Spezifikation.
- Unterstützung beim Entwurf von Benutzeroberflächen mit dem Canvas-Tool.
- Ermöglicht den Zugriff auf einzelne Elemente der Benutzeroberfläche zur Laufzeit der Anwendung.

Die Hauptaufgabe eines Builder ist die Erzeugung der Benutzeroberfläche. Zur Erzeugung eines Fenster und der darin enthaltenen grafischen Komponenten benötigt der Builder ein Spezifikations-Objekt der Klasse `FullSpec`. Dieses Objekt besitzt die zwei Exemplarvariablen `window` und `component`, die jeweils ein Objekt von der Klasse `WindowSpec` und `SpecCollection` enthalten. Das `WindowSpec`-Objekt beschreibt den Aufbau eines Fensters und enthält zum Beispiel Angaben bezüglich der Fenstergröße, der Fensterfarbe und des Fenstertitels. Das `SpecCollection`-Objekt hält eine Liste mit `ComponentSpec` vor, die, wie in Kapitel 2.2.2 erläutert, die einzelnen grafischen Komponenten eines Fensters beschreiben. Da ein `FullSpec`-Objekt nur ein einziges `WindowSpec`-Objekt referenzieren kann, besitzt jedes Fenster in VisualWorks® seinen eigenen Builder.

Die Erzeugung einer Oberfläche geschieht jedoch nicht direkt vom Builder selbst, sondern wird an die jeweilige `ComponentSpec` delegiert. Der Builder ruft die Methode

```
addTo: builder withPolicy: policy
```

auf, die jede `ComponentSpec`-Klasse von `UISpecification` erbt. Die Standardimplementierung durch `UISpecification` sieht wie folgt aus:

```
addTo: builder withPolicy: policy

    self dispatchTo: builder withPolicy: aPolicy.
    self finalizeComponentIn: builder
```

Die `ComponentSpec` delegiert die Erzeugung einer grafischen Komponente weiter an das angegebene `Policy`-Objekt, denn die Implementierung von `dispatchTo:` lautet:

```
dispatchTo: builder withPolicy: aPolicy

      policy window: self into: builder
```

Erst das konkrete `Policy`-Objekt erzeugt alle benötigten Objekte wie Wrapper, Views und das Model und initialisiert die gesamte Komponente. Diese Vorgehensweise hat den Vorteil, daß unterschiedliche `Policy`-Objekte unterschiedliche Darstellungsweisen ein und derselben Komponente ermöglichen. In VisualWorks® gibt es für jedes unterstützte Betriebssystem eine eigene Policy-Klasse. Einen kleinen Nachteil stellt diese Lösung für die Entwicklung von eigenen grafischen Komponenten wie den Netz-IAT dar. Damit eigene Komponenten überhaupt dargestellt werden, reicht es nicht, die entsprechenden Klassen in das System zu laden, sondern es müssen auch Änderungen an bestehenden Systemklassen bzw. Objekten des bereits laufenden Smalltalk-Systems gemacht werden.

3.4 Das Canvas-Tool

Das Canvas-Tool ist Bestandteil des VisualWorks®-System und unterliegt als solches bereits den technischen Prinzipien nach denen in diesem System Oberflächen erstellt werden. Nachdem wir in dem vorangegangenen Abschnitt bereits die Aufgabe eines Builders innerhalb von VisualWorks® beschrieben haben, möchten wir nun zeigen, welche Schritte, ausgehend von dem VisualLauncher, notwendig sind, um das Canvas-Tool zu starten. In diesem Zusammenhang ist es wichtig, das Canvas-Tool als eigenständige Anwendung zu sehen, die aus dem VisualWorks®-System heraus gestartet wird.

Innerhalb der Klasse `VisualLauncher` existiert in der Kategorie `actions` eine Methode `toolsNewCanvas`, die genau dann aufgerufen wird, wenn der Canvas-Button in dem Launcher Fenster gedrückt wurde. Innerhalb dieser Methode wird dann mit der folgenden Code-Zeile ein neues Objekt vom Typ `UIPainter` erzeugt, an dem dann die Methode `openNewWindowCanvas` aufgerufen wird.

```
UIPainter new openNewWindowCanvas
```

Die Methode `openNewWindowCanvas` findet man in der Klasse `UIPainter` unter der Kategorie `interface opening`. Durch den Aufruf von `setupBuilder` und `openPainterWindows`: erreicht diese Methode, daß auf dem Monitor die in Abbildung 2.6 dargestellte Anwendung erscheint.

```
OpenNewWindowCanvas

      self setupBuilder.
      Self openPainterWindows: builder spec
```

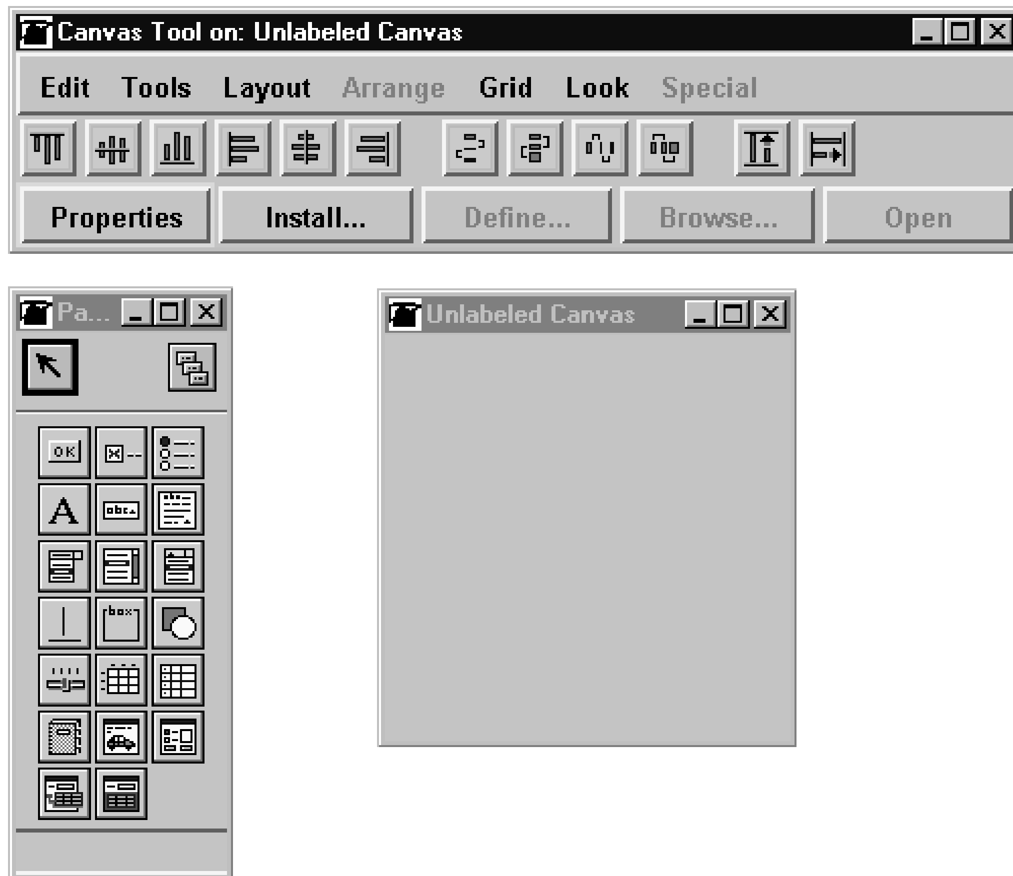


Abbildung 2.6

Die Klasse `UIPainter` ist eine Unterklasse von `ApplicationModel`, so daß in einem Objekt vom Typ `UIPainter` auf eine Exemplarvariable `builder` vom Typ `UIBuilder` zurückgegriffen werden kann. Mit Hilfe der Methode `setupBuilder` wird dieser Builder nun initialisiert. Zu dieser Initialisierung gehört zum Beispiel das Eintragen einer View, eines Controllers, eines Models und einer Window-Specification. Die hier versorgte Exemplarvariable `builder` und die dazugehörige `Specification` ist die Grundlage für das Fenster, welches in Abbildung 2.6 mit „Unlabeled Canvas“ betitelt ist. Die Methode `setupBuilder` ist in der Klasse `UIPainter` in der Kategorie `private` implementiert.

Verfolgt man den Kontrollfluß innerhalb der Methode `setupBuilder`, so erkennt man, daß über den Aufruf von `windowOn: aModel` (Klasse: `UIBuilder` / Kategorie: `building windows`) die Erstellung der Window-Specification durchgeführt wird. Die Methode `windowOn: aModel` erhält als Parameter für `aModel` das Objekt vom Typ `UIPainter` und setzt den Methodenaufruf wie folgt um:

```
self windowOn: aModel label: WindowSpec initialLabel
```

Hinter dem Aufruf initialLabel verbirgt sich eine konstante Funktion der Klasse WindowSpec, die den Fenstertitel „Unlabeled Canvas“ als String zurückliefert. Insgesamt besehen ruft hier das Builder-Objekt, welches mit dem UIPainter am Anfang erzeugt wurde, die Methode windowOn: label: (Klasse: UIBuilder / Kategorie: building windows) an sich selbst auf. Die Aufgabe der gerufenen Methode besteht darin, eine WindowSpec zu erzeugen, diese mit den notwendigen Werten zu versorgen und dann an den zuständigen UIBuilder weiterzureichen. Die Erweiterung der Exemplarvariable builder erfolgt dann über den Aufruf der Methode aSpec addTo: self withPolicy: policy (Klasse: UIBuilder / Kategorie: building), deren Aufgaben wir bereits im Abschnitt 2.3 beschrieben haben. Für den weiteren Ablauf im Hinblick auf die Versorgung eines UIBuilders setzen wir auf den im vorangegangenen Abschnitt beschriebenen Schritten auf.

Zum jetzigen Zeitpunkt haben wir allerdings erst die WindowSpec für ein Fenster erzeugt und uns fehlen noch die beiden Fenster, die uns die Funktionalität zur Erstellung von Oberflächen zur Verfügung stellen. Zudem ist noch keines der Fenster auf dem Bildschirm sichtbar. Für die Darstellung aller Fenster sorgt der Aufruf der Methode

```
self openPainterWindows: builder spec,
```

welche aus der bereits beschriebenen Methode openNewWindowCanvas erfolgt. Die hierüber erzeugten Fenster sind abhängig von dem Zustand des „Unlabeled Canvas“, daß heißt, daß wenn zum Beispiel das „Unlabeled Canvas“-Fenster geschlossen wird, ebenfalls das Tools- und das Paletten-Fenster geschlossen werden. Die Methode openPainterWindows findet man in der Klasse UIPainter in der Kategorie private.

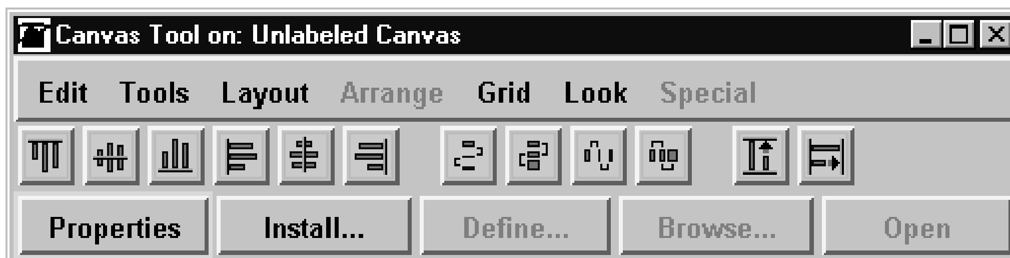


Abbildung 2.7

Nachdem über die Methode setupBuilder die Exemplarvariable builder mit einer WindowSpec initialisiert worden ist, wird diese nun in der openPainterWindows:-Methode ausgelesen, um ein Fenster zu erzeugen. Das eigentliche Öffnen des Fensters erfolgt über den Aufruf der Methode aWindow openWithExtent: aPoint andType: aSymbol. Nachdem nun das Fenster mit dem Namen „Unlabeled Canvas“ erzeugt wurde, wird es mit der Methode beMaster zum übergeordneten Fenster für alle anderen Fenster erklärt, die zur selben Anwendung gehören.

Mit dem Aufruf von `self openToolsFor: mainWindow` schließlich wird das in Abbildung 2.7 dargestellte Fenster geöffnet. In diesem Fenster sind alle Funktionen angeordnet, mit denen man zum Beispiel Widgets innerhalb eines Fensters gruppieren kann. Für die genaue Beschreibung der Arbeitsweise mit dem Canvas-Tools sei auf [1] und [2] verwiesen.

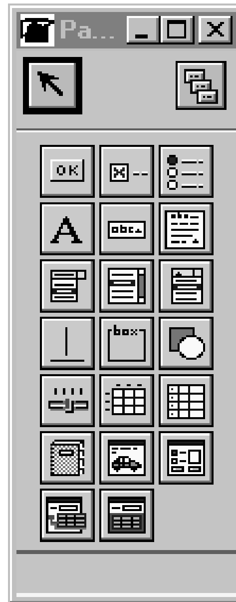


Abbildung 2.8

Als letztes wird das Palettenfenster geöffnet, welches die einzelnen für die Gestaltung eines Fensters verwendbaren Elemente wie zum Beispiel Buttons und Edit-Felder enthält. Die Erweiterung des Fensterinhaltes im Hinblick auf unseren Netz-IAT werden wir im Kapitel 4 näher betrachten. Das eigentliche Öffnen des Palettenfensters erfolgt mit dem Aufruf von `self openPaletteFor: mainWindow` aus der Methode `openPainterWindows:` heraus. Das so erzeugte Fenster ist noch einmal in der vorstehenden Abbildung 2.8 dargestellt.

Mit dem Aufruf von `accept` am Ende der Methode `openPainterWindows`: ist der Start des Canvas-Tools beendet. Da für den Hochlauf des Canvas-Tools mehrere Schritte notwendig sind, haben wir mit dem in der Abbildung 2.9 dargestellten Interaktionsdiagramm versucht, diesen Vorgang zu verdeutlichen.

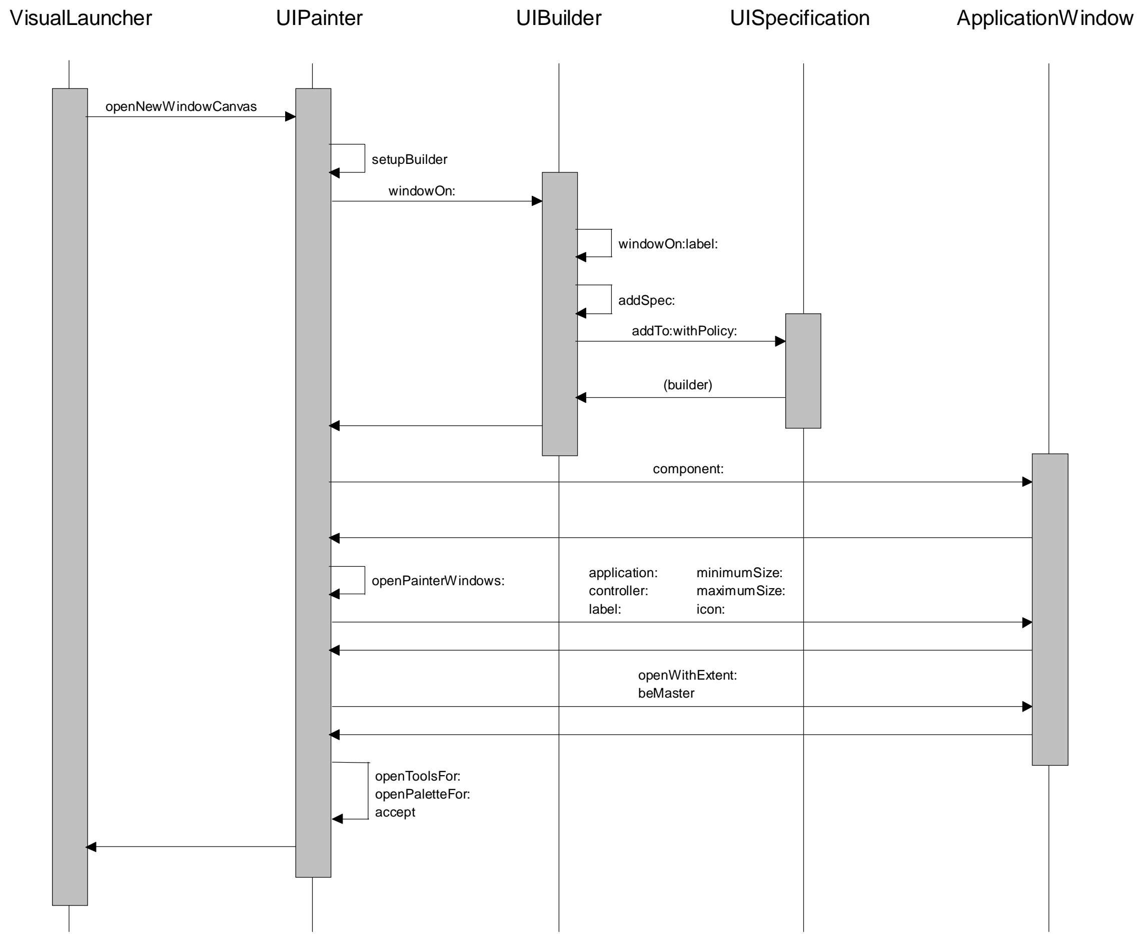


Abbildung 2.9

Verwendet man das Canvas-Tool schließlich, um zum Beispiel ein oder mehrere Fenster für eine bestimmte Anwendung zu erstellen, so werden diese Informationen in Form einer `FullSpec` in einer vom Benutzer zu bestimmenden Klasse hinterlegt. Man kann die so erstellte Specification auch manuell editieren. Hierfür wählt man für die entsprechende Klasse die Klassensicht und geht in die Kategorie `interface specs`. In dieser Kategorie findet man eine Methode mit dem Namen `windowSpec`, in der sich der Aufbau des Fensters als `literalArray`[3] verbirgt.

[3] Ein Literal ist eine Art von Smalltalk-Code, der ein Objekt beschreibt. Der Interpreter kann zur Laufzeit aus einem Literal das Objekt erzeugen, das es beschreibt.

4 Entwurf eines Interaktionstypen zur Bearbeitung von Netzen

Dieser 3 Abschnitt gliedert sich in zwei Teile. Innerhalb der Kapitel 3.1 und 3.2 werden wir auf die Spezifikation unseres Netz-Interaktionstypen eingehen, während wir im Kapitel 3.3 die Implementierung des IAT innerhalb des Smalltalk Systems VisualWorks® 2.5 auszugsweise beschreiben.

4.1 Anforderungen an den Interaktionstypen

Bei den hier beschriebenen Anforderungen des Interaktionstypen handelt es sich zum einen um das Verhalten, das der IAT zur Laufzeit zeigen soll, und zum anderen um die Art, wie der IAT in das Smalltalk-System eingebettet sein soll.

Die erste Forderung, die wir an unseren IAT stellen, besteht darin, daß diese Konstruktion Funktionalität im Hinblick auf das VisualWorks® Fenstersystem gegenüber der diesen IAT verwendenden Software kapseln soll. Da jede Anwendung nur über die definierte Schnittstelle auf den IAT zugreifen kann, ist es dann leichter möglich, die vorhandene Software an andere Fenstersysteme anzupassen.

Im Anschluß an den vorgenannten Aspekt stellen wir die Anforderung, daß ein Netz-IAT ein grafisches Oberflächenelement sein soll. Dies bedeutet, daß während der Erstellung von Oberflächen bzw. eines Dialoges für eine Anwendung mit Hilfe des Canvas-Tools auf einen Netz-IAT genauso zugegriffen werden kann, wie beispielsweise auf ein Edit-Feld oder eine List-Box. Desweiteren sollen die Eigenschaften des Interaktionstypen sofort nach der Plazierung dieses Widgets in einem Fenster, dem sogenannten Properties-Tool, einstellbar sein. Der IAT soll die Darstellung des ihm übergebenen Netzes selbst bestimmen. Die Festlegung der Art der Darstellung erfolgt zum Zeitpunkt der Erstellung der Oberfläche und kann zur Laufzeit nicht mehr geändert werden.

Bei der Benutzung von Interaktionstypen ist zu jedem Zeitpunkt ersichtlich, welche Werte durch ihn dargestellt werden. Für unseren Interaktionstypen ist es also erforderlich, daß ein fachlicher Wert „Netz“ modelliert wird, der dem Netz-IAT übergeben wird. Die Art der Darstellung obliegt dann, wie bereits erwähnt, dem Interaktionstypen selbst. Durch die Einführung des fachlichen Wert „Netz“ erreichen wir, daß die Schnittstelle zwischen der Oberfläche und der darunter liegenden Software erweitert wird. Durch die eindeutige Zuordnung eines Interaktionstypen zu einem Wert, die bereits bei der Erstellung der Oberfläche vorgenommen wird, erreichen wir zudem eine Typsicherheit, da der Interaktionstyp anhand des Typs des übergebenen Wertes entscheiden kann, ob es sich um einen gültigen oder ungültigen Wert handelt.

Desweiteren wollen wir erreichen, daß der IAT kontextunabhängig ist. Dieses bedeutet, daß die Darstellung und Bearbeitung des Netzes ausschließlich durch die voreingestellten Eigenschaften des Interaktionstypen selbst bestimmt wird, nicht aber durch Zustände der eigentlichen Anwendung beeinflußt werden kann.

Die Handhabung unseres Netz-IAT soll letztendlich einen funktionalen Charakter haben. Für uns bedeutet dieses, daß man dem IAT zur Laufzeit einen fachlichen Wert vom Typ Netz übergeben kann, woraufhin dieser mit Hilfe der Funktionen des IAT bearbeitet wird und schließlich an die Anwendung als fachlicher Wert vom Typ Netz zurückgegeben wird. Auf die Zustände, die der IAT während der Bearbeitung des Netzes hat, darf nicht zugegriffen werden.

4.2 Funktionalität des Netz-IAT

Nachdem wir in dem vorangegangenen Abschnitt 3.1 die Forderungen formuliert haben, die wir an das Konzept eines Interaktionstypen für Netze stellen, möchten wir nun diejenigen Ziele definieren, die sich mit der Funktionalität des Netz-IAT auseinandersetzen.

Zunächst einmal soll es einfach sein, ein Netz zu erstellen und zu bearbeiten. Für die Erstellung und Manipulation eines Netzes sollen die bei grafischen Oberflächen üblichen Hilfsmittel wie zum Beispiel Maus und Menüs verwendet werden. Im weiteren soll der Netz-IAT nicht auf eine bestimmte Netz-Art ausgerichtet sein. Dieses fordert eine generelle Implementierung eines Netzes als ein Tupel A = (N,E), wobei N die Knoten des Netzes und E die Kanten repräsentiert. Zusätzlich soll es möglich sein, das zur Laufzeit erstellte bzw. an den Interaktionstyp als fachlichen Wert übergebene Netz mit Hilfe einer Grammatik zu überprüfen. Die dem Netz zugrunde liegende Grammatik wird bereits bei der Festlegung der Eigenschaften des IAT im Properties-Tool angegeben. Auf die mit Hilfe des Netz-IAT erstellten und / oder bearbeiteten Netze muß die den IAT verwendende Software zur Laufzeit Zugriff haben. Das heißt, die Anwendung kann an dem IAT eine Methode aufrufen, die das aktuelle Netz zurückliefert.

Da das VisualWorks®-System dieser Implementierung zugrunde liegt, möchten wir im weiteren noch auf die Zielsetzung für die Handhabung des Interaktionstypen innerhalb dieser Entwicklungsumgebung eingehen. Um den IAT als ein Element der Oberfläche verwenden zu können, müssen wir das Canvas-Tool, welches wir in Abschnitt 2.4 beschrieben haben, dahingehend erweitern, daß der Netz-IAT in der Tool-Bar erscheint und somit für die Entwicklung von Oberflächen verwendbar ist. Hierin eingeschlossen ist das Bereitstellen von Methoden, mit deren Hilfe die Eigenschaften des Netz-IAT's wie zum Beispiel Farbe, Darstellungsform und Grammatik vorgenommen werden können. Um Vermischungen zwischen der Darstellung eines Netzes und den Bestandteilen eines Netzes zu vermeiden, wollen wir eine getrennte Modellierung des Netzes - als einen fachlichen Wert - und seiner grafischen Information vornehmen. Zu der grafischen Information eines Netzes gehören einzig die Koordinaten der Netzknoten. Diese sind aber nicht Bestandteil der fachlichen Modellierung eines Netzes. Aus diesem Grunde muß die Anwendung die Koordinaten der Knoten auch gesondert von dem Netz-IAT anfordern können. Um verschiedene Sichten auf ein Netz gewährleisten zu können, darf keine Zuordnung zwischen einer dem Netz zugrunde liegenden Grammatik und der gewählten Form der Darstellung erfolgen. Somit ist es möglich, zum Beispiel in einem Fenster der Oberfläche zwei Netz-IAT's zu plazieren, denen aber unterschiedliche Darstellungsformen bei gleicher Grammatik zugeordnet worden sind.

Gleiches gilt im Bezug auf die Grammatiken. Für die Modellierung der Grammatiken und der Darstellungsformen haben wir auf abstrakte Klassen zurückgegriffen, die bereits die Schnittstelle definieren, die der Netz-IAT bei der Verwendung benötigt. Unser Ziel war hierbei die Möglichkeit auf einfache Art und Weise den IAT um neue Grammatiken und Darstellungen zu erweitern, ohne daß an den grundlegenden Bestandteilen des Netz-IAT Veränderungen vorgenommen werden müssen.

4.3 Implementierung des Interaktionstypen

In diesem Abschnitt werden wir den Entwurf und die Besonderheiten der Implementierung des IAT's beschreiben. Den an Details interessierten Leser verweisen wir auf die im Anhang angegebene Web-Seite, über die der gesamten kommentierten Quellcode verfügbar ist.

4.3.1 Die Funktion der Klasse Net

Das Model für den Netz-IAT ist ein beliebiges Netz, bestehend aus einer Menge von Knoten und einer Menge von Tupeln, die gerichtete Kanten zwischen den Knoten beschreiben. Dieses Netz wird mit Hilfe der Klassen `Net`, `Node`, `GraphicalNode` und `Edge` modelliert.

Die Klasse `Net` verwaltet das gesamte Netz des IAT und stellt damit das Model für die `NetView` dar. Intern werden die Knoten- und Kantenobjekte in `Sets` gespeichert. Aus zwei Gründen wurde das Netz nicht in einer Matrix gespeichert. Der erste Grund besteht darin, daß VisualWorks® keine zweidimensionalen Matrizencontainer zur Verfügung stellt, deren Zeilen- und Spaltenanzahl sich zur Laufzeit dynamisch verändern läßt. Wir haben darauf verzichtet, eine solche Matrix selber zu entwerfen, da sich Netze, die mit Hilfe von grafischen Tools bearbeitet werden sollen, in der Regel nicht sehr viele Kanten besitzen. Ansonsten wäre eine grafische Erstellung und Bearbeitung kaum sinnvoll zu bewerkstelligen. Die Verwendung von `Sets` stellt somit einen Kompromiß zwischen einem einfachen Entwurf und Geschwindigkeitsaspekten dar.

Die Klasse `Net` stellt eine Reihe von Methoden zum Verändern und Sondieren des Netzes zur Verfügung. Es können zum Beispiel Knoten oder Kanten eingefügt und gelöscht werden, oder es kann überprüft werden, ob zwei Knoten miteinander durch eine Kante verbunden sind. Diese Methoden stellen wir hier nicht weiter vor, da sie zum großen Teil sehr kurz und einfach zu verstehen sind.

Da es sich bei dem Netz um ein Model handelt, muß das Netzobjekt beobachtbar sein und alle seine Beobachter, bei unserem IAT die `NetView`, über Zustandsänderungen informieren. Da die Klasse `Object` in VisualWorks® bereits alle benötigten Methoden für einen Beobachtermechanismus besitzt und alle Klassen Unterklassen von `Object` sind, mußte kein eigener Beobachtermechanismus implementiert werden. Alle Objekte in VisualWorks® sind immer beobachtbar oder können selber andere Objekte beobachten. Bei einer Zustandsveränderung benachrichtigt ein `Net`-Objekt seine View durch den Aufruf der Methode

```
self changed: aSymbol with: aParameter,
```

wobei das Symbol und der Parameter je nach Zustandsänderung unterschiedlich sind.

Die Knoten des Netzes werden durch Objekte vom Typ `Node` repräsentiert. Ein Knoten besitzt eine Bezeichnung und einen Typ. Für diesen Zweck besitzt die Klasse `Node` die zwei

Exemplarvariablen `type` und `label`, die über sondierende und verändernde Methoden ausgelesen und gesetzt werden können. Für die grafische Darstellung eines Netzes wird auch die Position eines Knoten in Bezug zu den anderen Knoten benötigt. Deshalb benutzt der Netz-IAT die von `Node` abgeleitete Klasse `GraphicalNode`. Eine `GraphicalNode` besitzt zusätzlich die Exemplarvariable `origin`, die die Koordinaten eines Knoten in Form eines `Point` speichert.

Bei Veränderungen des Zustandes eines Knoten werden, wie beim Netz, alle Beobachter benachrichtigt. Beobachter eines Knoten ist in dem Netz-IAT nicht das Netzobjekt, sondern eine `NodeView`. Jeder Knoten und jede Kante hat ihre eigene View, die für die entsprechende Darstellung verantwortlich ist. Ein Knoten bzw. Kante ist somit Bestandteil eines Netzes und gleichzeitig Model für eine eigene View.

Die Kanten des Netzes werden durch Objekte der Klasse `Edge` repräsentiert. Eine Kante besitzt eine Bezeichnung, einen Typ, einen Startknoten und einen Endknoten, die jeweils in Exemplarvariablen gespeichert werden. Alle Kanten in dem Netz-IAT sind gerichtete Kanten, denn es wird explizit zwischen Start- und Endkonten unterschieden. Falls in einem Netz ungerichtete Kanten benötigt werden, so ist darauf zu achten, daß die verwendete Grammatik und die verwendeten Kantenviews nicht zwischen Start- und Endknoten differenzieren.

Es gibt im Gegensatz zur `GraphicalNode` keine Klasse GraphicalEdge, da die Koordinaten des Start- und Endpunktes einer Kante von der jeweiligen View des Start- bzw. Endknoten bestimmt werden. Wie schon weiter oben erwähnt, besitzt auch jede Kante ihre eigene View, die bei Zustandsänderungen benachrichtigt wird.

4.3.2 Die Klasse NetGrammar

Die Klasse `NetGrammar` ist eine rein abstrakte Basisklasse für die syntaktische Überprüfung von Netzen. Sie beschreibt nur die Schnittstelle, die der Netz-IAT zur Erzeugung und Überprüfung von Netzen benötigt. Die folgenden drei Methoden müssen von konkreten Unterklassen implementiert werden. Die Methode `returnNodeTypes` muß eine `OrderedCollection` mit den Symbolnamen aller im Netz erlaubten Knotentypen zurückliefern. Gleiches gilt für die Methode `returnEdgeTypes`, die jedoch alle erlaubten Kantentypen zurückliefert. Zur syntaktischen Prüfung des Netzes muß ein Grammatikobjekt die Nachricht `isValid: aNet` verstehen. Der Rückgabewert dieser Methoden muß entweder `true` oder `false` sein. Es bietet sich an, falls die verwendete Art der Grammatik dieses erlaubt, diese Überprüfung nach dem Verhaltensmuster „Interpreter“ zu implementieren. Eine Beschreibung dieses Entwurfmusters ist in [5] zu finden.

4.3.3 Die Klasse NetRepresentation

Auch die Klasse `NetRepresentation` ist eine abstrakte Basisklasse, die nur die benötigte Schnittstelle beschreibt. Die `NetRepresentation` hat die Aufgabe, eine Abbildung zwischen den Knoten- und Kantentypen im Netz und den zu verwendenden Views vorzunehmen. Dieses ermöglicht der `NetView`, zu jedem Knoten und jeder Kante die passende Viewklasse zu finden und ein entsprechendes Objekt davon zur Darstellung zu erzeugen. Die einzige Nachricht, die eine konkrete Unterklasse von `NetRepresentation` verstehen muß, ist `returnViewClassfor: aType`. Diese Funktion muß für jedes mögliche Typsymbol der verwendeten Grammatik den Namen der anzuwendenden Viewklasse zurückliefern. Mit Hilfe dieses Klassennamens kann die `NetView` dann das benötigte Objekt erzeugen. Zweckmäßigerweise wird während der Initialisierung des Objektes vom Typ `NetRepresentation` ein `Dictionary` angelegt, welches die Tupel (#Typ, Viewklasse) aufnimmt. Dieses `Dictionary` kann dann in der `returnViewClassfor:` Methode abgefragt werden.

4.3.4 Die View-Klassen

Die grafische Darstellung des Netz-IAT auf dem Bildschirm erfolgt mit Hilfe der drei Klassen `NetView`, `NodeView` und `EdgeView`. Wobei die letzten beiden Klassen abstrakt sind und für die konkrete Darstellung eines bestimmten Netzes spezifiziert werden müssen. Die Klasse `NetView` hat zwei wichtige Aufgaben. Als erstes ist sie für die Darstellung des IAT und die Koordination aller Views zuständig. Die zweite Aufgabe ist die Bereitstellung der Schnittstelle zwischen dem Netz-IAT und der Anwendung. Im folgenden werden wir die von uns erstellten und benutzten View-Klassen vorstellen.

4.3.4.1 NetView

Die Darstellung des Netzes mit den Knoten und Kanten erfolgt nicht direkt durch die `NetView` selbst, sondern jeder Knoten und jede Kante hat seine eigene View, die nur für die Darstellung eines Knoten bzw. einer Kante zuständig ist. Auf diese Weise läßt sich der Netz-IAT sehr leicht und flexibel an unterschiedliche Darstellungsweisen von Netzen anpassen. Wenn zum Beispiel eine andere Darstellungsweise für einen bestimmten Knotentyp erwünscht ist, muß nur eine kleine `NodeView`-Klasse spezialisiert und ausgetauscht werden und nicht die gesamte `NetView` modifiziert werden. Die `NetView` delegiert und koordiniert alle an der Darstellung des Netzes beteiligten Views. Zu diesem Zweck verwaltet die `NetView` alle Views getrennt nach Knoten-Views und Kanten-Views in den Exemplarvariablen `node-Components` und `edgeComponents`. `nodeComponents` und `edgeComponents` sind vom Typ `CompositePart` und enthalten die jeweiligen Views in `TranslatingWrapper` verpackt.

Die Hauptaufgabe einer jeden View ist die Darstellung ihres Models. Zu diesem Zweck versteht jede View in VisualWorks die Methode displayOn: aGraphicsContext. Mit dem Aufruf dieser Methode wird eine View veranlaßt, ihr Model in dem angegebenen GraphicsContext zu zeichnen. Die NetView delegiert diese Aufgabe an die beiden CompositeParts, die die Knoten- und Kanten-Views enthalten. Ein CompositePart gibt diesen Aufruf weiter an alle enthaltenen Wrapper und Views. Die displayOn: Methode der NetView ist deshalb sehr kurz:

```
displayOn: aGraphicsContext

  aGraphicsContext paint: self backgroundColor.
  aGraphicsContext displayRectangle: self bounds.
  aGraphicsContext paint: self foregroundColor.
  self edgeComponents displayOn: aGraphicsContext.
  self nodeComponents displayOn: aGraphicsContext.
```

Als erstes wird die Zeichenfarbe gesetzt und dann der Hintergrund durch Überzeichnen mit einem Rechteck gelöscht. Anschließend werden alle Kanten und dann alle Knoten in den übergebenen GraphicsContext gezeichnet.

Zur Bearbeitung des Netzes muß der Benutzer Knoten und Kanten selektieren können. Dies geschieht durch Anklicken der jeweiligen grafischen Repräsentation eines Knoten bzw. einer Kante. Der NetController ermittelt die Position eines Mausklicks und fragt dann bei der NetView nach, ob und welche View an der Position liegt. Die NetView prüft in der Methode hitDetect: aPoint, ob der übergebene Punkt innerhalb der Grenzen einer der von ihr verwalteten View liegt. Dazu ruft sie an den beiden CompositeParts nodeComponent und edgeComponent auch die Methode hitDetect: aPoint auf, die alle CompositeParts verstehen. Das Ergebnis dieses Methodenaufrufes ist entweder nil, wenn sich keine View an der angefragten Position befand oder der Translating-Wrapper der View, die sich an der Position befindet. Über den Aufruf der Methode toggleNodeSelection: aView bzw. toggleEdgeSelection: aView der Net-View kann der NetController den Selektionszustand des Netzes verändern.

Jede Node- und EdgeView unterscheidet zwei Zustände, den selektierten Zustand und den nicht selektierten Zustand. Diese beiden Zustände unterscheiden sich jedoch nur in der Form der grafischen Darstellung. In den Exemplarvariablen selectedNodes und selected-Edges vom Typ OrderedCollection verwaltet die NetView alle Views, die sich im selektierten Zustand befinden. Durch Aufruf der Methoden deselectAllNodePatterns, deselectAllEdgePatterns, selectAllNodePatterns, selectAll-EdgePatterns, toggleNodeSelection und toggleEdgeSelection kann der NetController den Zustand der Selektion des Netzes verändern. Als sondierende Methoden stehen die Funktionen selectedNodes und selectedEdges zur Verfügung.

Die Darstellung des Netzes durch die NetView erfordert es, daß die NetView über alle Veränderungen des Netzes informiert wird. Deshalb trägt sich die NetView als Beobachter des Netzes beim Net ein. Diese Funktionalität erbt die NetView von ihrer Oberklasse View und braucht deshalb nicht selbst implementiert werden. Es sei an dieser Stelle nochmals betont, daß das Netz das Model der NetView ist und von ihr beobachtet wird; jedoch die einzelnen Knoten und Kanten des Netzes nicht von der NetView beobachtet werden, sondern Models ihrer eigenen Views sind und nur von dieser beobachtet werden. Das bedeutet, daß im Falle der Zustandsänderung eines der Knoten nicht die NetView benachrichtigt wird, sondern die zugehörige NodeView. Die NetView wird immer dann benachrichtigt, wenn Knoten und Kanten zum Netz hinzugefügt oder gelöscht werden. Dies geschieht im Rahmen des Benachrichtigungsmechanismus von VisualWorks® durch den Aufruf der Methode update: aSymbol with: anObject. Die Aufgabe der NetView ist es, die entsprechenden Views zu den Knoten bzw. Kanten zu erzeugen oder zu löschen. Die folgende Tabelle 3.1 gibt an, auf welche Nachrichtensymbole die NetView reagiert und welche privaten Methoden die update: Routine zur Bearbeitung aufruft.

Symbol	Methodenaufruf
#addNode	self addNode: anObject
#removeNode	self removeNode: anObject
#addEdge	self addEdge: anObject
#removeEdge	self removeEdge: anObject

Tabelle 3.1

Als Objekt wird jeweils der Knoten oder die Kante mit übergeben, die hinzugefügt oder gelöscht wurde. Dies ist notwendig, da die NetView ansonsten nicht in der Lage ist, zu erkennen, was sich am Netz geändert hat. Am Beispiel der Methode addNode: soll im folgenden das Hinzufügen einer NodeView erklärt werden. Der Aufruf von addNode: erfolgt ausschließlich durch die NetView selbst und nur dann, wenn sie von ihrem Model die Nachricht #addNode empfangen hat. Als Parameter erhält die NetView den Knoten vom Typ GraphicalNode, der dem Netz durch den NetController hinzugefügt worden ist. Als erstes holt sich die NetView den Typ des Knotens und läßt sich dann von der eingestellten NetRepresentation die zugehörige NodeView geben. Bei der erhaltenen NodeView trägt sie den Knoten als Model ein und setzt den zu verwendenden Textstyle. Der letzte Schritt ist das Hinzufügen der neuen NodeView zu den nodeComponents. Dieses geschieht durch den Aufruf der Methode add: aView at: anOrigin an den nodeComponents. Dieser Aufruf fügt nicht direkt die neue NodeView hinzu, sondern er fügt einen TranslatingWrapper der die NodeView enthält hinzu, damit die NodeView an der richtigen Stelle erscheint. Die Koordinaten für den TranslatingWrapper und damit auch der NodeView gibt die GraphicalNode vor. Nach dem Hinzufügen der beiden Komponenten veranlaßt das CompositePart automatisch eine Neuzeichnung der enthaltenen Views.

Das Hinzufügen einer neuen `EdgeView` läuft im wesentlichen nach dem gleichen Schema ab. Der einzige Unterschied ist die etwas kompliziertere Berechnung des Start- und Endpunktes einer Kante. Da eine Kante in unserem Entwurf keine Koordinaten enthält, muß die `NetView` die benötigten Koordinaten für den Start- und Endpunkt der Kante sowie die Koordinaten für den benötigten `TranslatingWrapper` erst anhand der Koordinaten der beiden verbundenen Knoten berechnen. Die `NetView` versucht dabei immer, die Länge einer Kante zu minimieren.

Da es bei der Darstellung eines Knoten im Netz-IAT keine einschränkenden Vorgaben gibt und diese nur von der verwendeten `NodeView` abhängt, kann die `NetView` die zur Darstellung der Knoten passenden Start- und Endpunkte nicht alleine bestimmen. Sie bestimmt anhand der Position und Ausdehnung der verbundenen Knoten mit Hilfe der Methode `findDirections: firstRectangle to: secondRectangle` nur den groben Verlauf der Kante. Diese Methode liefert zwei Symbole aus der Menge `{#Left, #Right, #Top, #Bottom}` zurück, die angeben, wo der gesuchte Punkt in Bezug zum Knoten liegt. Jede `NodeView` liefert auf Anfrage mit einem der vorher genannten Symbole als Parameter einen sogenannten Ankerpunkt zurück, der zur Darstellung des jeweiligen Knoten paßt und als Start- oder Endpunkt einer Kante dient. Anhand dieser Angaben kann die `NetView` die `EdgeView` mit den benötigten Koordinaten initialisieren und zu den `edgeComponents` hinzufügen.

Das Entfernen eines Knoten ist dagegen relativ einfach zu implementieren. Die Methode `removeNode: aGraphicalNode` erhält als Parameter den aus dem Netz zu entfernenden Knoten. Zunächst wird in den `nodeComponents` der `TranslatingWrapper` zu der View gesucht, die den Knoten als Model führt. Als nächstes wird geprüft, ob die View selektiert ist, wenn dieses der Fall ist, wird sie zunächst aus der Liste der selektierten `Node-Views` entfernt. Danach wird der Wrapper und die View aus den `nodeComponents` entfernt. Als letztes ruft die `NetView` an sich selbst `changePreferredBounds` auf, da sich durch das Entfernen eines Knoten evtl. die Ausdehnung des dargestellten Netzes geändert haben könnte. Das Entfernen einer `EdgeView` läuft analog zum Entfernen einer `NodeView`.

Eine View in VisualWorks® wird nicht nur benachrichtigt, wenn sich ihr Model verändert hat, sondern es werden auch Nachrichten nach dem Entwurfsmuster „Zuständigkeitskette“ (Chain of Responsibility) erläutert in [5] in der Hierarchie der `VisualParts` an sie weitergeleitet. Dieser Mechanismus wird auch von der `NetView` genutzt. Veränderungen an den Knoten und Kanten werden nur der jeweiligen `Node-` bzw. `EdgeView` mitgeteilt. Sie können jedoch indirekt für die `NetView` von Interesse sein. Wenn zum Beispiel die Position eines Knoten durch den Benutzer verändert wurde, dann muß dies der `NetView` von der betroffenen `NodeView` mitgeteilt werden, damit die `NetView` die Koordinaten des zugehörigen `TranslatingWrapper` anpassen kann.

Die Methode upcastEvent: aKey with: aParamter from: anInitiator path: aPathCollection, die jede View versteht, gibt das Ereignis aKey in der Hierarchie der VisualParts weiter. Die NetView redefiniert diese Methode, um die Nachrichten #NodeViewChangedBounds, #NodeViewChangedOrigin und #EdgeViewChangedBounds, die von einer Node- oder EdgeView gesendet werden können, abzufangen. Als Parameter werden von unserem Netz-IAT nur aKey und anInitiator benutzt, wobei anInitiator das Senderobjekt der Nachricht angibt.

Die Nachricht #NodeViewChangedBounds bedeutet, daß sich die Ausmaße einer NodeView verändert haben. Dies ist für die NetView in dem Fall von Bedeutung, wenn sich die Ausmaße verkleinert haben. Eine NodeView kann sich nur innerhalb ihrer aktuellen Ausmaße neuzeichnen. Wenn sich die Ausmaße durch Veränderung der Beschriftung eines Knoten verkleinern, dann ist es für die NodeView unmöglich, den nun überstehenden Bereich zu löschen, da er nicht mehr innerhalb ihrer neuen Ausmaße liegt. Die NodeView sendet deshalb die Nachricht #NodeViewChangedBounds aus und die NetView reagiert auf diese Nachricht mit zwei Aktionen. Als erstes paßt sie Koordinaten aller Kanten, die von dem betroffenen Knoten ausgehen, an seine neuen grafischen Ausmaße an und anschließend zeichnet sie den Netz-IAT komplett neu.

Die Nachricht #NodeViewChangedOrigin sagt aus, daß ein Knoten verschoben wurde und nun eine neue Position einnimmt. Dies erfordert von der NetView neben dem Anpassen der betroffenen Kanten auch ein Verschieben des TranslatingWrapper der entsprechenden NodeView. In diesem Fall ist nur ein Neuzeichnen aller veränderten Views notwendig.

Die Veränderung der Beschriftung einer Kante veranlaßt die zuständige EdgeView, die Nachricht #EdgeViewChangedBounds zu senden. Auf diese Nachricht reagiert die NetView mit dem Neuzeichnen des gesamten Netz-IAT durch den Aufruf der Methode self invalidate.

Neben der Darstellung des Netzes ist es auch die Aufgabe der NetView, eine Schnittstelle zwischen der Anwendung und dem Netz-IAT anzubieten. Alle Methoden dieser Schnittstelle befinden sich in der Kategorie application accessing. Die beiden wichtigsten Methoden sind getNet und setNet: getNet liefert eine tiefe Kopie des aktuell bearbeiteten Netzes und setNet: setzt ein neues Netz zur Bearbeitung. setNet: setzt dabei eine tiefe Kopie des übergebenen Netzes als neues Model der NetView ein. Dies erfordert auch eine Redefinition der Methode model:, da entsprechend des neuen Netzes alle zugehörigen Node- und EdgeViews mit den schon beschriebenen Methoden erzeugt werden müssen.

Zum Abschluß der Beschreibung des Aufbaus der NetView soll noch erklärt werden, wie die Zuordnung zwischen der NetView und dem NetController in VisualWorks® realisiert worden ist. Die Funktion defaultControllerClass einer jeden View in VisualWorks® liefert das Klassenobjekt des zu verwendenden Controllers zurück. Im Rahmen der Erzeugung des Netz-IAT wird entsprechend dem Ergebnis des Funktionsaufrufes an der NetView das passende Controllerobjekt erzeugt und mit der NetView verbunden. Die NetView liefert immer die Klasse NetController als Standardcontroller zurück.

4.3.4.2 NodeView

Die NodeView ist die abstrakte Basisklasse für alle Views, die Knoten im Netz darstellen. Sie stellt fast die gesamte Funktionalität zur Verfügung, die Views zur Darstellung von Knoten im Netz-IAT benötigen. Es müssen im einfachsten Fall nur die drei Methoden displayNormalOn:, displaySelectedOn: und preferredBounds implementiert werden. Die ersten beiden Methoden müssen die Darstellung eines Knoten in den übergebenen GraphicsContext zeichnen. Die Methode displayNormalOn: muß einen Knoten im nicht selektierten Zustand und die Methode displaySelectedOn: im selektierten Zustand zeichnen.

In der Exemplarvariable selected speichert eine NodeView als booleschen Wert, ob sie selektiert ist oder nicht. Dementsprechend ist die Standard-Darstellungsmethode displayOn: der NodeView implementiert. Wenn selected den Wert true besitzt, wird nach displaySelectedOn: verzweigt, ansonsten nach displayNormalOn:. Über die Methode selected kann der Selektionszustand abgefragt werden und über die Methode toggleSelection kann dieser Zustand verändert werden.

Die Methode preferredBounds muß immer die Ausmaße der grafischen Darstellung als Rectangle-Objekt zurückgeben. Sie muß also passend zu den beiden Display-Methoden implementiert werden. Diese Methode wird von den Views bzw. Wrappern, die in der Hierarchie der grafischen Komponenten oberhalb der jeweiligen NodeView liegen, zum selektiven Neuzeichnen oder zur Hitdetection benutzt.

Als weitere Exemplarvariable hält jede NodeView displayLabel vom Typ ComposedText. Über die Methode displayLabel können die Display-Methoden der NodeView auf einen fertig initialisierten ComposedText zugreifen, der die Knotenbeschriftung enthält und in einem GraphicsContext darstellbar ist. Die Methode displayLabel ruft, falls die Exemplarvariable displayLabel noch nicht initialisiert ist, die Methode createDisplayLabel auf, die die Variable initialisiert. Ansonsten liefert sie nur den Wert der Exemplarvariablen zurück. Diese Vorgangsweise hat den Vorteil, daß das Objekt vom Typ ComposedText nur einmal erzeugt werden muß. Dies führt zu einer besseren Performanz, da das displayLabel bei jedem Neuzeichnen benötigt wird.

Da die Klasse ComposedText eine Unterklasse von VisualComponent ist, besitzt sie auch die displayOn: at: Methode und kann einfach in einem GraphicsContext an einer beliebigen Position dargestellt werden. Der Vorteil eines ComposedText ist, daß dieser verschiedene Formatierungen und Schriftarten verwenden kann. Standardmäßig wird der Fontstyle verwendet, der bei der Initialisierung durch die NetView gesetzt wird und in der Exemplarvariablen textStyle gespeichert wird.

Die Methode returnAnchorPoint: einer NodeView muß, wie bereits in Abschnitt 3.3.4.1 erwähnt, je nach übergebenem Symbol einen zur Darstellung passenden Ankerpunkt zurückgeben. Ein Ankerpunkt dient dabei als Start- und Endpunkt einer Kante. Die Standardimplementierung der Klasse NodeView gibt als Ankerpunkt jeweils die Mitte der äußeren Kante der Ausmaße der grafischen Darstellung zurück, wobei die Ausmaße anhand der preferredBounds bestimmt werden. Diese Methode muß unter Umständen reimplementiert werden, wenn die zurückgegebenen Ankerpunkte nicht zur Darstellung passen. Dieses könnte zum Beispiel der Fall sein, wenn die Knoten als Dreiecke dargestellt werden sollen.

Wie jede View, muß auch die NodeView auf Änderungen ihres Model mit einer Anpassung der grafischen Darstellung reagieren. Da ein Knoten nur die Nachrichten #setLabel und #setOrigin versendet, muß die Methode update: aSymbol nur diese beiden Fälle behandeln. Die Nachricht #setLabel bedeutet, daß sich die Beschriftung eines Knoten verändert hat. Die NodeView reagiert auf diese Nachricht mit drei Aktionen. Als erstes wird der ComposedText in der Exemplarvariablen displaylabel durch Aufruf von self createDisplayLabel angepaßt. Da dies in der Regel eine Änderung der Ausmaße zur Folge hat, ruft die NodeView an sich selber changedPreferredBounds: nil auf. Diese Methode, die jedes VisualPart versteht, benachrichtigt alle abhängigen Objekte über die Änderung der Ausmaße. Als Parameter können die alten Ausmaße oder nil übergeben werden. Der Wert nil bedeutet, daß die alten Ausmaße nicht mehr bekannt sind. Als letztes teilt die NodeView die Änderung der Ausmaße über die Methode updateEvent: allen interessierten Objekten in der Hierarchie der grafischen Komponenten des Netz-IAT mit. Dies ist notwendig, da der TranslatingWrapper, der jeder NodeView vorangestellt ist, die erste Nachricht der Methode changedPreferredBounds: nicht weiterleitet, wenn sich die Ausmaße verkleinert haben. Die NetView benötigt jedoch diese Nachricht um das Neuzeichnen des Netz-IAT zu veranlassen. Nähere Informationen zu diesem Nachrichten-mechanismus stehen im vorherigen Kapitel 3.3.4.1 bei den Erklärung zur NetView.

Als letztes soll die Implementierung der Methode defaultControllerClass beschrieben werden, da hierin eine Besonderheit enthalten ist. Eine NodeView hat keinen eigenen Node-Controller, da der NetController alle Benutzereingaben entgegennimmt und Veränderungen am Netz veranlaßt. Da aber jede View in VisualWorks® einen Controller besitzen muß, liefert die Methode defaultControllerClass als Klassennamen Controller zurück. Dies ist die Basisklasse aller Controller in VisualWorks®. Sie stellt zwar die gesamte Funktionalität zur Bearbeitung von Benutzereingaben zur Verfügung, reagiert aber selbst nicht auf die Eingaben. Statt dessen wird die Bearbeitung von Benutzer-eingaben an andere übergeordnete Controller weitergeleitet.

4.3.4.3 EdgeView

Der Aufbau und die Methoden der Klasse EdgeView entsprechen im wesentlichen denen der NodeView. Auch die EdgeView ist eine abstrakte Klasse, deren Display-Methoden noch nicht implementiert sind.

Neben den bei der NodeView bereits vorhandenen Exemplarvariablen selected, displayLabel und textStyle besitzt eine EdgeView zusätzlich die zwei Variablen startPoint und endPoint. In diesen beiden Variablen speichert die EdgeView die Koordinaten des Start- und Endpunkte der darzustellenden Kante. Diese Koordinaten werden von der NetView berechnet und über die Methoden setStartPoint und setEndPoint gesetzt. Zum Sondieren der Koordinaten gibt es die Methoden returnStartPoint und returnEndPoint.

Der einzige erwähnenswerte Unterschied zwischen der Klasse EdgeView und der Klasse NodeView liegt in den Methoden returnAnchorpoint und containsPoint:. Die zuerst genannte Methode besitzt eine EdgeView nicht, da eine Kante die Ankerpunkte zwischen zwei Knoten verbindet und nicht Kanten mit Kanten. Die Nachricht containsPoint:, die jede VisualComponent versteht, wurde bei der EdgeView redefiniert. Die ursprüngliche Implementierung in der Klasse VisualComponent prüft, ob der übergebene Punkt innerhalb der Ausmaße (preferredBounds) der View liegt. Wenn ja, liefert die Methode true zurück. Diese wird aus der Methode hitDetection: heraus aufgerufen, das heißt containsPoint: wird benutzt, um festzustellen, ob eine View durch den Benutzer angeklickt worden ist oder nicht. Bei der Klasse NodeView bereitet die ursprüngliche Implementierung keine Probleme, da bei den Subklassen von NodeView die grafische Repräsentation in der Regel die gesamte Fläche der View gleichmäßig abdeckt. Bei der Darstellung von Kanten ist dies meistens anders. Wenn zum Beispiel eine Kante diagonal durch die rechteckigen Ausmaße einer EdgeView verläuft, dann nimmt die grafische Darstellung der Kante nur einen minimalen Bruchteil der Gesamtfläche ein. Würde in einem solchen Fall die ursprüngliche Implementierung verwendet werden, könnte der Benutzer weit entfernt von der Kante in eine Ecke der Ausmaße der EdgeView klicken und diese so selektieren. Eine Kante sollte aber sinnvollerweise nur selektiert werden, wenn der Benutzer ungefähr in die Nähe der gezeichneten Kante klickt. Deshalb wurde containsPoint: so implementiert, daß nur true zurückgegeben wird, wenn der Abstand zwischen dem übergebenen Punkt und der Kantenlinie einen bestimmten Mindestabstand nicht überschreitet. Die konstante Funktion maxDistance liefert hierzu den geforderten Mindestabstand.

Ansonsten verhalten sich alle Routinen der EdgeView ähnlich wie die Routinen der NodeView bzw. besitzen zum großen Teil die gleichen Implementierungen.

4.3.5 Der Net-Controller

Der NetController im Netz-IAT hat die Aufgabe, Benutzereingaben zu erfassen und daraufhin Veränderungen an dem mit der View gemeinsam benutzten Model vorzunehmen. Die Klasse NetController erbt ihre Basisfunktionalität von der Klasse ControllerWithMenu und diese wiederum erbt von der Klasse Controller. Die Klasse ControllerWithMenu wurde deshalb verwendet, weil sie bereits in der Lage ist, mit Kontextmenüs umzugehen. Diese Menüs werden zur Anzeige von Befehlen verwendet.

Um zu verstehen, welche Methoden des NetController zur Auswertung der Benutzerinteraktionen implementiert bzw. redefiniert werden mußten, folgt zunächst eine Beschreibung über den Ablauf des Kontrollflusses in VisualWorks®. Details sind in [1 zu finden.

Nicht nur Objekte vom Typ View, sondern auch alle Objekte vom Typ ScheduledWindow besitzen einen Controller. Bei einem ScheduledWindow ist dies ein StandardSystemController. Die StandardSystemController aller Fenster sind einem Objekt bekannt, das in der systemglobalen Variablen ScheduledControllers gespeichert ist. Die Aufgabe von ScheduledControllers besteht darin, reihum bei allen bekannten Objekten vom Typ StandardSystemController durch Senden der Nachricht isControllWanted nachzufragen, ob der Empfänger der Nachricht die Kontrolle übernehmen will.
Dies ist der Fall, wenn der StandardSystemController mit true antwortet. Diese Antwort liefert ein StandardSystemController dann, wenn Änderungen am Fensterinhalt ausgegeben werden müssen oder wenn das ScheduledWindow das aktive Fenster des Fenstersystems ist. Wenn ScheduledControllers einen StandardSystemController entdeckt hat, der die Kontrolle übernehmen will, sendet er diesem die Nachricht startUp. Hierauf reagieren alle Controller im wesentlichen gleich. Zuerst senden sie sich selbst die Nachricht controlInitialize. In dieser Methode kann der Controller Initialisierungen vornehmen. Danach schickt sich der Controller die Nachricht controlLoop, die für die Verarbeitung von Eingaben und anderen Ereignissen sorgt, bis der Controller die Kontrolle abgibt. Nach Ausführung von controlLoop führt der Controller noch die Methode controlTerminate aus. Dort können noch abschließende Aktionen ausgeführt werden, bevor der Controller die Kontrolle wieder an den Sender der Nachricht startUp zurückgibt. Innerhalb der Methode controlLoop ruft der Controller an sich selbst isControlActive auf. Solange dieser Aufruf true liefert, wird die Methode controlActivity aufgerufen. Die controlActivity Methode des StandardSystemControllers prüft zunächst, ob der mittlere Button der Maus gedrückt ist. Wenn das nicht der Fall ist, geht er davon aus, daß er nicht für weitere Aktionen zuständig ist. Er versucht dann, die Kontrolle an einen Controller abzugeben, der zu einer View in der Hierarchie von Grafikobjekten seines Fensters gehört. Dazu sendet er dem ScheduledWindow die Nachricht subViewWantingControl. Das Fenster fragt daraufhin seine component mit der Nachricht objectWantingControl nach einer View, dessen Controller daran interessiert ist, die Kontrolle zu übernehmen. Diese Nachricht wird von allen Wrappern und CompositeParts in der Hierarchie der Grafikobjekte nur weitergereicht ohne sie zu beachten. Wenn die Nachricht eine View ohne Controller erreicht, sendet diese nil zurück. Eine View mit Controller sendet die Nachricht isControlWanted an ihren Controller. Antwortet dieser mit true, dann gibt die View sich selbst als Ergebnis zurück, andernfalls ist das Resultat wiederum nil. Der StandardSystemController erhält also entweder nil oder eine View als Ergebnis der Ausführung. Wenn das Ergebnis nil ist, beendet der StandardSystemController seine Aktivitäten. Bekommt er dagegen eine View, dann schickt er dieser die Nachricht startUp. Die View gibt die Nachricht startUp an ihren Controller weiter, der darauf die bereits bekannte Folge von Initialisierung (controlInitialize), Kontrollschleife (controlLoop) und abschließenden Maßnahmen (controlTerminate) durchläuft. Innerhalb der controlLoop wird, wenn der Aufruf von isControlActive true zurückgeliefert hat, die Methode controlActivity gerufen. controlActivity ist

nun die Methode, in der die eigentliche Reaktion auf die Eingabe erfolgt. Die Abbildung 3.2 gibt den Kontrollfluß nochmals als Interaktionsdiagramm wieder.

In der Klasse `NetController` wurde nur die Methode `controlActivity` redefiniert, da keine Initialisierung und keine abschließenden Aktionen ausgeführt werden müssen. Die Aufgabe dieser Methode im `NetController` ist es, nur zu prüfen, ob die Delete-Taste gedrückt wurde. Das Drücken dieser Taste soll alle selektierten Elemente löschen. Die Implementierung hierfür ist sehr kurz und kompakt:

```
controlActivity

  self sensor keyboardPressed
       ifTrue: [self processKey: self sensor keyboardEvent].
  super controlActivity.
```

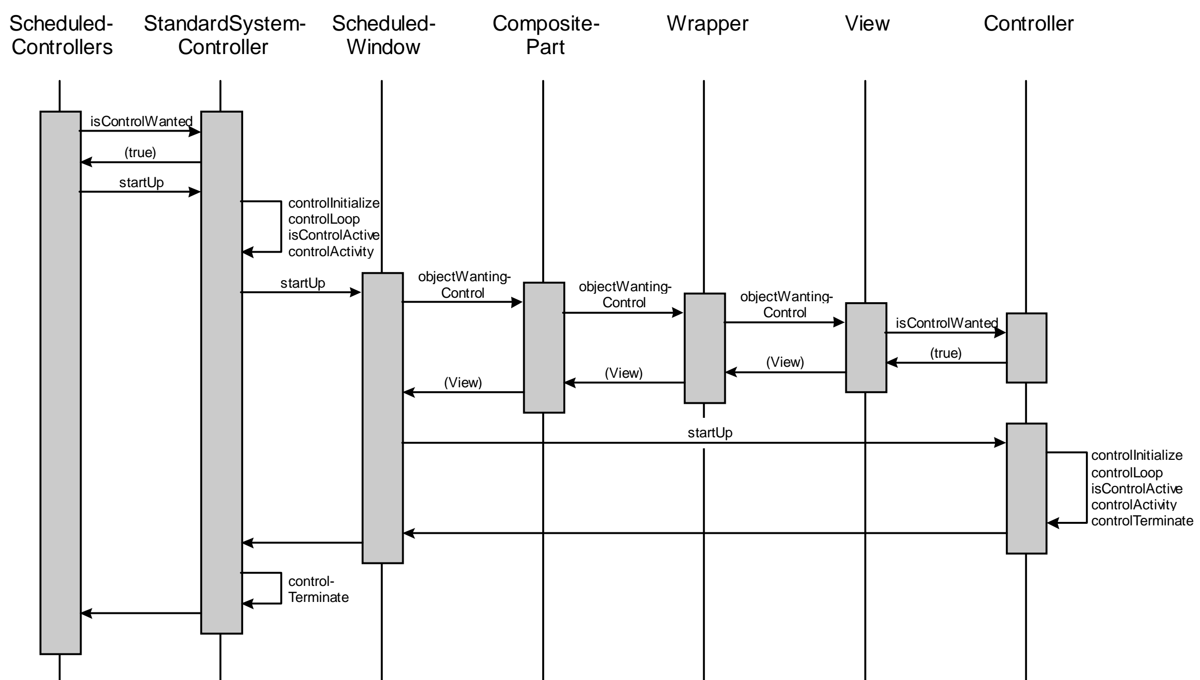


Abbildung 3.2

Die Methode benutzt einen `Sensor`, den jeder Controller besitzt. Dieser `Sensor` gibt Auskunft über die Position der Maus, den Zustand der Mausknöpfe, auf der Tastatur eingegebene Zeichen und darüber, ob eine Sondertaste (Shift-, Control-, Meta- oder Alternate-Tase) gedrückt ist. Wenn eine Taste gedrückt ist, dann wird zur Methode `processKey:` verzweigt. Diese Methode bearbeitet in unserem Netz-IAT alle Tastatureingaben. Momentan prüft sie nur, ob die Delete-Taste gedrückt ist und ruft, falls das der Fall ist, `remove-Selection` auf. Diese Methode entfernt dann die selektierten Elemente aus dem Netz.

Wichtig ist, daß die ursprüngliche Implementierung der Oberklasse `ControllerWithMenu` ausgeführt wird, weil diese unter anderem die Maustasten prüft und mit entsprechenden Methodenaufrufen reagiert.

Die Methode `redButtonActivity` wird bei einem `ControllerWithMenu` immer dann aufgerufen, wenn der Benutzer die linke Maustaste gedrückt hat. Der `NetController` führt

in dieser Methode mit Hilfe der NetView ein hitDectect: aus, um zu prüfen, ob ein Grafikelement angeklickt worden ist. Die Methode hitDectect: liefert entweder den TranslatingWrapper der entsprechenden View oder nil, wenn kein Element an der Position lag zurück. Über den TranslatingWrapper besorgt sich der NetController die zugehörige View und verzweigt je nach dem Typ der View entweder zu redButton-NodeActivity oder zu redButtonEdgeActivity. Die Methode redButtonNode-Activity prüft zunächst, ob der Benutzer den Knoten verschieben möchte. Dazu wird die Methode moveComponent gerufen, die mit Hilfe des Sensors die Mausbewegungen überwacht und die Position der View anpaßt. Die Methode liefert nach dem Loslassen des linken Mausknopfes die neue Position zurück. Falls sich die neue und die alte Position um mehr als zwei Pixel unterscheiden, geht die Methode redButtonNodeActivity davon aus, daß der Benutzer den Knoten verschieben wollte, und paßt die Koordinaten der Graphical-Node (das Model der NodeView) an. Wenn dies nicht der Fall ist, dann wird nur die Selektion in Abhängigkeit des Zustandes der Shift-Taste geändert. Die Methode red-ButtonEdgeActivity ändert nur den Selektionszustand der entsprechenden Kante, da Kanten vom Benutzer nicht verschoben werden können.

Da ein ControllerWithMenu bereits die Funktionalität zur Darstellung eines Kontextmenüs besitzt, müssen in der Klasse NetController nur zwei Methoden reimplementiert werden. In der Exemplarvariablen menuHolder wird, durch einen ValueHolder gekapselt, das eigentliche Menü von Typ Menu gehalten. Während der Initialisierung eines Exemplars von ControllerWithMenu wird die Methode setMenu gerufen. Diese Methode erzeugt in der Regel das Menüobjekt, kapselt es in einem ValueHolder und speichert diesen in der Exemplarvariablen menuHolder. Wenn der Benutzer nun zur Laufzeit den rechten Mausknopf drückt, wird aus der Methode controlActivity der Oberklasse ControllerWithMenu heraus die Methode menu aufgerufen. In diese Methode wird nur das Menüobjekt aus der Exemplarvariablen menuHolder zurückgeliefert. Der Sinn dieser Methode ist es, daß der NetController hier die Möglichkeit besitzt, vor der Anzeige des Menüs einzelne Menüeinträge zu verändern. Der NetController verändert in dieser Methode einen Teil der Menüeinträge so, daß nur zum Selektionszustand passende Aktionen anwählbar sind. Als weitere Aktion vervollständigt er bei jedem Aufruf die Vorgabemenüs, die die bereits benutzen Knoten- und Kantenbeschriftungen enthalten.

Der Aufbau eines Kontextmenüs ist relativ einfach. In der Methode setMenu wird zunächst ein Exemplar der Klasse Menu erzeugt. Zu diesem Exemplar können ausschließlich über die Methode addItem: einzelne Menüeinträge oder über die Methode addItemGroup: ein ganzes Array von Einträgen hinzugefügt werden. Wenn die Methode addItemGroup: benutzt wird, dann werden die so hinzugefügten Gruppen von Menüeinträgen bei der Anzeige des Menüs durch eine waagerechte Linie voneinander abgetrennt. Die einzelnen Menüeinträge sind jeweils Exemplare der Klasse MenuItem. Über die Methode label: bzw. labeled: während der Initialisierung kann der Text des Menüpunktes gesetzt werden. Die Verbindung des Menüeintrages mit einem Methodenaufruf geschieht durch den Aufruf der Methode value: am MenuItem. Dieser Methode wird entweder der Methodenname, der nach Anwahl des Menüpunktes aufgerufen werden soll, als Symbol übergeben oder es wird ein Block übergeben, der Zum Zeitpunkt der Auswahl ausgeführt wird. Die erste Möglichkeit ist aber nur für Methodenaufrufe geeignet, die keine Parameter erwarten. Um parametrisierte Methoden aufzurufen, muß ein entsprechender Block übergeben werden, der dann den eigentlichen Aufruf vornimmt. Wenn z.B. ein neuer Knoten in das Netz eingefügt werden soll, wird die Methode newNode: type des NetController gerufen. Der Typ des Knoten

wird dabei als Parameter übergeben. Um nun diese Methode durch Anwahl eines entsprechenden Menüeintrages aufzurufen, müssen entsprechende Blöcke generiert werden.

Die folgende Implementierung wird in `setMenu` verwendet:

```
nodeTypes := self view getNetGrammer returnNodeTypes.
nodeBlocks := nodeTypes asArray collect:
                [:type | [:dummy | self newNode: type]].
```

Der Parameter `dummy` muß verwendet werden, da `MenuItem` dem Block immer einen Parameter mitgibt, der jedoch hier nicht benötigt wird.

Für die Erzeugung von Untermenüs wird an einem Exemplar der Klasse `MenuItem` die Methode `subMenu:` gerufen und ein Objekt vom Typ `Menu` als Parameter mitgegeben. Das Kontextmenü des Netz-IAT wird so mit den beschriebenen Methoden erzeugt und dann in einem `ValueHolder` gekapselt in der Exemplarvariablen `menuHolder` gespeichert.

Um später in der Methode `menu` Menüeinträge zu verändern, wird die Methode `menu-ItemLabeled:` zum Auffinden von Menüeinträgen verwendet. Diese Methode erwartet den Text des zu suchenden Menüeintrages und liefert, falls vorhanden, das passende Objekt vom Typ `MenuItem` zurück. Diesem Objekt können dann Nachrichten wie `enable`, `disable`, `beOn` oder `beOff` geschickt werden. Die Methode `menu` paßt so die Menüeinträge dem internen Zustand des Netz-IAT an.

4.3.6 Die NetSpecification

Die Specification für unseren Netz-IAT liefert die Klasse `NetSpec`. Ein Objekt dieser Klasse speichert die Informationen, die benötigt werden, um einen Netz-IAT zu erzeugen. Während der Entwicklung der Oberfläche einer Anwendung wird bei der Benutzung des Netz-IAT unter anderem ein Objekt dieser Klasse angelegt. Alle Einstellungen des Netz-IAT, die im Properties-Fenster vorgenommen werden, werden in diesem Exemplar gespeichert. VisualWorks® speichert nun nicht die Oberflächenobjekte selbst, sondern serialisiert die Exemplare der Spezifikationsklassen in Form von `literalArrays`.

Die Klasse `NetSpec`, eine Unterklasse von `WidgetSpec`, erweitert für den Netz-IAT die Standardeigenschaften eines `Widgets` in VisualWorks® um die Attribute Grammar, Layout und Fontstyle. Diese besonderen Eigenschaften eines Netz-IAT werden bereits zur Entwicklungszeit der Oberfläche im Properties-Fenster festgelegt und in den Exemplarvariablen `netGrammar`, `netRepresentation` und `style` eines Exemplars der `NetSpec` gespeichert. Damit diese Eigenschaften über das Properties-Fenster editierbar sind, muß zu jedem Attribut eine Lese- und eine Schreibmethode definiert werden. Diese beiden Methoden müssen den gleichen Methodennamen besitzen, damit sie durch einen `AspectAdaptor` angesprochen werden können. Sie unterscheiden sich nur dadurch, daß die Schreibmethode einen Parameter erwartet. Die Signaturen der Methoden zum Lesen und Schreiben einer Netzgrammatik lauten zum Beispiel `netGrammar und netGrammar: anObject`.

Für das Editieren der neuen Attribute müssen die entsprechenden Dialoge im Properties-Tool erweitert werden. Die neuen Attribute haben wir bei unserem Netz-IAT auf den Seiten „Basics“ und „Details“ des Properties-Fensters untergebracht. Um dieses zu erreichen, wurden

die entsprechenden Ressourcen aus der Oberklasse `WidgetSpec` in das Canvas-Tool geladen, dort geändert und die geänderten Ressourcen in der Klasse `NetSpec` abgespeichert. Die serialisierten Ressourcen des „Basic"-Dialoges sind in der Methode `basicEditSpec` und die Ressourcen des „Details"-Dialoges in der Methode `detailsEditSpec` gespeichert.

Damit das Properties-Tool zur Laufzeit weiß, welche Dialoge angezeigt werden sollen und wo die entsprechenden Ressourcen dafür zu finden sind, antwortet jedes Exemplar von `NetSpec` auf die Nachricht `slices` mit einem `literalArray`, welches die entsprechenden Literale der Exemplare der Spezifikationsklassen der Dialoge enthält. Anhand dieser Literale können zunächst die Exemplare der Spezifikationsklassen und mit deren Hilfe die eigentlichen Dialoge erstellt werden. Da für den Netz-IAT nicht alle Attribute, die über die Dialoge der Klasse `WidgetSpec` zur Verfügung gestellt werden, benötigt werden, wurde die Methode `slices` so redefiniert, daß nur noch die Dialoge „Basic", „Details", „Color" und Position im `literalArray` enthalten sind und angezeigt werden.

Die Verbindung der Eingabefelder für die Attribute mit den entsprechenden Lese- und Schreibmethoden erfolgt über `AspectAdaptor`. Zu diesem Zweck müssen bei den neu hinzugefügten Eingabefeldern im Properties-Fenster eindeutige Aspektsymbole gesetzt werden. Die eigentliche Erzeugung und Verbindung der `AspectAdaptor` mit den entsprechenden Methoden geschieht in der Methode `addBindingsTo: env for: inst: channel: aChannel` in einem Objekt vom Typ `NetSpec`. Die erste Anweisung in der redefinierten Methode sollte immer

```
super addBindingsTo: env for: inst channel: aChannel
```

lauten. Auf diese Weise wird sichergestellt, daß alle Attribute der Oberklasse `WidgetSpec` ebenfalls editierbar sind. Die Verbindung des Eingabefeldes für die Netzgrammatik mit den Methoden `netGrammar` und `netGrammar:` wird mit der folgenden Anweisung erreicht

```
env at: #netGrammar put:
                        (self adapt: inst
                        forAspect: #netGrammar
                        channel: aChannel)
```

Hierbei stellt `env` die vom Builder übergebene `bindings`-Variable dar. Die Variable `inst` ist zur Laufzeit die `NetSpec` selbst. Sie dient als Zielobjekt des `AspectAdaptor`, daß heißt an der `NetSpec` ruft der `AspectAdaptor` die jeweiligen Methoden auf. `aChannel` ist ein `ValueHolder`, der auch das Objekt vom Typ `NetSpec` enthält. Nähere Informationen über `AspectAdaptor` und ihre Anwendung sind in [2] zu finden.

Die neue Komponente benötigt neben den angepaßten Dialogen auch noch zwei Icons und einen Namen. Die Ressourcen für ein farbiges und ein schwarz/weißes Icon werden wiederum in Form eines `literalArray` in den Methoden `paletteIcon` und `paletteMonoIcon` gespeichert. Den Namen der Komponente erhält das Canvas-Tool durch den Aufruf der Methode `componentName`, die den String „Net-IAT" zurückliefert.

Neben den bereits besprochenen Methoden wurden noch drei weitere Methoden der Oberklasse `WidgetSpec` redefiniert. Die Methode `defaultFlags` liefert als Ergebnis ein Bitfeld, bestehend aus vier Bits zurück. Diese vier Bits bestimmen, ob die Komponente einen Rahmen, ein Menü, einen horizontalen Srollbalken und einen vertikalen Scrollbalken besitzt.

Da der Netz-IAT alle diese Eigenschaften besitzen soll, liefert die Methode `defaultFlags` ein Bitfeld zurück, bei dem alle vier Bits gesetzt sind. Die Methode `defaultModel` muß eine Exemplar des Models des Netz-IAT zurückliefern. Dieses Model und die Objekte aller im Netz-IAT verwendeten Klassen werden schon zur Entwurfszeit der Oberfläche benötigt, da sich der Netz-IAT zu diesem Zeitpunkt in dem Entwurfsfenster des Canvas-Tools darstellen muß. In unserem Fall liefert die Methode `defaultModel` ein Objekt vom Typ `Net` zurück. Die letzte Methode, die von uns redefiniert wurde, heißt `dispatchTo: policy with: builder`. Diese Methode wird im Rahmen des Erzeugungsprozesses des Netz-IAT aufgerufen. Die Aufgabe dieser Methode besteht darin, den gesamten Netz-IAT mit allen Exemplaren zu erzeugen. Da jedoch viele Komponenten in ihrer Darstellung von dem verwendeten Fenstersystem (`UILookPolicy`) abhängig sind, delegieren die meisten Spezifikationsklassen den Prozeß der Erzeugung an das als Parameter übergebene `policy`-Objekt. Eine nähere Beschreibung des Erzeugungsprozesses und des Kontrollflusses ist im Kapitel 2.3 zu finden. Um die spätere Anpassbarkeit des Netz-IAT an unterschiedliche Fenster- und Darstellungssysteme zu erleichtern, delegiert auch die `NetSpec` die Erzeugung an das `policy`-Objekt. Die einzige Anweisung der Methode `dispatchTo:with:` lautet dementsprechend

```
policy net: self into: builder.
```

Die hierbei aufgerufene Methode `net:into:` ist ausführlich im Abschnitt 4.2 beschrieben.

4.3.7 Beziehungen innerhalb des Interaktionstypen

Wir möchten in diesem Teil der Arbeit die Benutzt-Beziehungen der im vorwege beschriebenen Klassen im Hinblick auf den gesamten Interaktionstypen zusammenfassen. Hierbei teilen wir den Netz-IAT in zwei Teile auf. Zum einen in den Teil, der die dem Netz zugrunde liegenden Klassen enthält und zum anderen in den Teil, der das eigentliche Netz beschreibt.

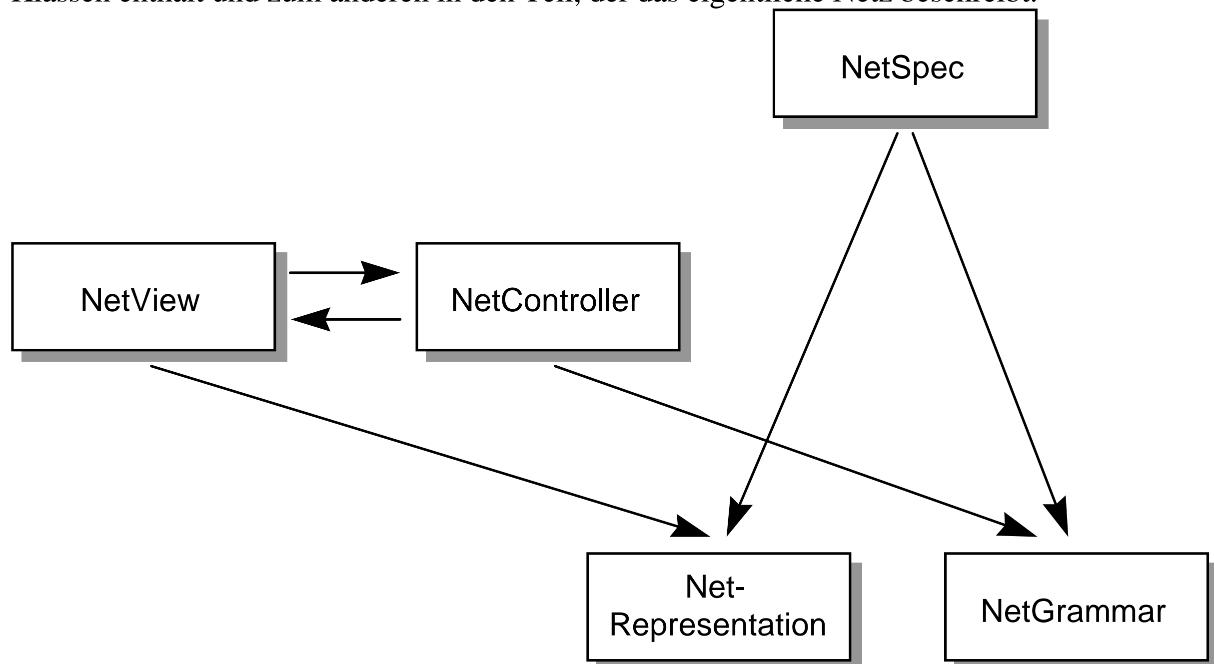


Abbildung 3.2

Die Namen der beiden zu verwendenden Unterklassen von `NetRepresentation` und `NetGrammar` werden von dem Benutzer in dem Properties-Fenster des Canvas-Tool versorgt. Zu dem Zeitpunkt, da aus einer `NetSpec` heraus das eigentliche Oberflächenelement erzeugt wird, müssen Objekte der zu den eingetragenen Namen korrespondierenden Klassen erzeugt werden, woraus die in Abbildung 3.2 eingetragene Beziehung resultiert. Der `NetController` hingegen benutzt die `NetGrammar` genau dann, wenn er durch die Auswahl des Menupunktes „Check Net“ dazu aufgefordert wird. In diesem Fall wird von dem `NetController` die Prüfmethode `isValid:` der `NetGrammar` aufgerufen. Für die `NetView` ergibt sich die Notwendigkeit auf die `NetRepresentation` zurückzugreifen, wenn eine neue Kante oder ein neuer Knoten in das Netz eingefügt werden soll. Dann muß die `NetView` in Abhängigkeit von dem im Menu ausgewählten Typ die entsprechende View-Klasse herausfinden. Zu diesem Zweck wird die `NetRepresentation` benutzt.

Analog zu der Abbildung 3.2 zeigt die Abbildung 3.3 die Benutzt-Beziehungen auf der Ebene des Netzes selbst. Die `NetView` und der `NetController` benutzen sich gegenseitig, um die Bereitstellung der Funktionalität zur Bearbeitung eines Netzes zu gewährleisten. So ruft zum Beispiel der `NetController` an der `NetView` eine Methode auf, die die gerade angewählten Knoten zurückliefert, wenn eine neue Kante eingefügt werden soll. Wird nun von dem `Net` an die `NetView` gemeldet, daß eine neue Kante eingefügt wurde, so erzeugt die `NetView` ein Objekt vom Typ `EdgeView`, die wiederum ein Objekt vom Typ `Edge` als Model besitzt. Äquivalent hierzu verhält sich die `NetView` im Bezug auf die Knoten. Das eigentliche Netz schließlich ist von uns in der Art und Weise abstrahiert worden, daß es nur aus Knoten und Kanten besteht. Aus diesem Grund benutzt das `Net` nur `GraphicalNode` und `Edge` zur Netzmodellierung. Es sei an dieser Stelle noch einmal betont, daß jeder Knoten und jede Kante eine eigene View (`NodeView` bzw. `EdgeView`) zur Darstellung besitzt.

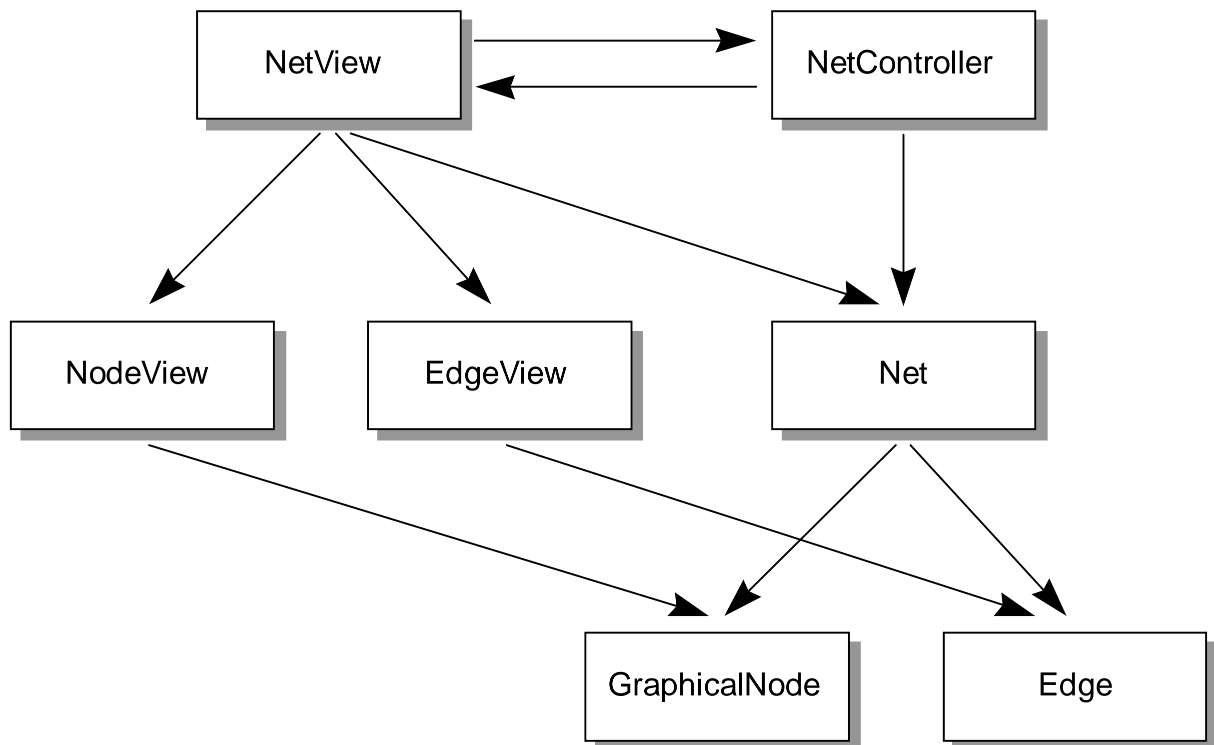


Abbildung 3.3

4.3.8 Das MVC-Paradigma angewandt auf den Netz-IAT

Nachdem wir in den vorhergehenden Abschnitten die Aufgaben der einzelnen Klassen innerhalb des Netz-IAT's beschrieben haben und im weiteren auf die Benutzt-Beziehungen zwischen diesen Klassen eingegangen sind, möchten wir an dieser Stelle kurz aufzeigen, wie sich das MVC-Paradigma in Bezug auf unseren Interaktionstypen darstellt.

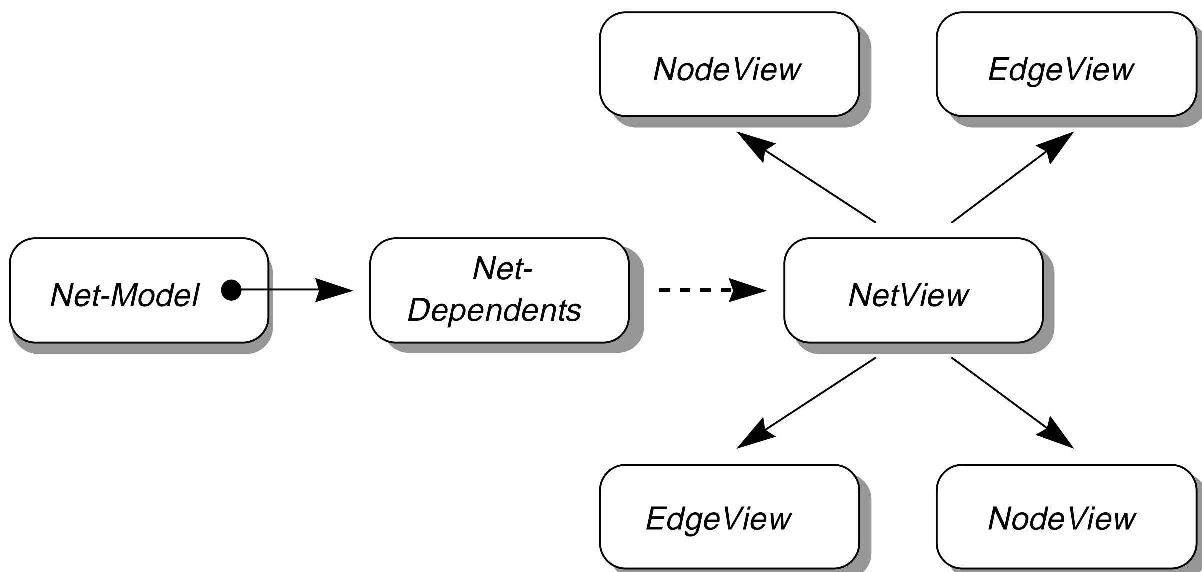


Abbildung 3.4

Zur Laufzeit eines Programmes, das einen Netz-IAT verwendet, reagiert der Interaktionstyp auf Änderungen des `Net`-Models, in dem die dort eingetragenen Dependents mit einer genau definierten Nachricht über die Art der Änderung informiert wird. In unserem Fall existiert für das `Net`-Model als einziger Dependent das Objekt vom Typ `NetView`, das zu dem IAT gehört. Innerhalb des `NetView`-Objektes wird dann der grafische Aufbau des Netzes koordiniert. Die genaue Abfolge der hierfür notwendigen Schritte haben wir bereits in Abschnitt 3.3.4.1 aufgeführt.

Die Struktur innerhalb des Interaktionstypen haben wir in Abbildung 3.4 auf der Ebene der `View`-Objekte dargestellt. Da wir für unseren Netz-IAT nur einen Controller verwenden, der von der `NetView` erzeugt wird und dieser direkt zugeordnet ist, haben wir diesen nicht dargestellt. Alle anderen Views - `NodeView` und `EdgeView` - haben in unserer Konstruktion keinen eigenen Controller. Im Vergleich zu der in Abbildung 2.3 dargestellten Beziehung zwischen Model, View und Controller innerhalb des MVC-Paradigmas, ist zu ersehen, daß wir die Konzepte des Paradigmas auf der Ebene der Net-Model, `NetView` und dem `NetController` einhalten. Desweiteren verletzt auch die Tatsache, daß ein Objekt vom Typ `NodeView` oder `EdgeView` keinen eigenen Controller besitzt, nicht das MVC-Paradigma.

4.3.9 Die Anwendungsschnittstelle des Interaktionstypen

In diesem Abschnitt möchten wir erläutern, welche Möglichkeit besteht, um auf unseren Netz-IAT zur Laufzeit eines Programmes zuzugreifen. Bevor wir konkret auf die vorgesehene Schnittstelle des Netz-IAT eingehen, werden wir aufzeigen, wie unser IAT in einem beliebigen `ApplicationModel` eingebunden ist, da innerhalb des VisualWorks®-Systems jede Benutzeroberfläche eine Unterklasse von `ApplicationModel` sein muß.

Wird nun eine grafische Oberfläche gestartet, in der auch ein Netz-IAT enthalten ist, so greift das jeweilige `ApplicationModel` während des Aufbaus der Variable `builder` auf die Methode `net:into:` der Klasse `UILookPolicy` zu. Über diese Methode wird der Zugriff auf die Net-Specification unseres Netz-Interaktionstypen geschaffen. Die Aufgaben der Klasse `NetSpec` und die dort hinterlegten Informationen haben wir bereits im Abschnitt 3.3.6 beschrieben. Desweiteren sorgt die Methode `net:into:` dafür, daß der Netz-IAT in den Builder des `ApplicationModels` der Oberfläche mit den entsprechenden Parametern eingefügt wird. Auf den Aufbau und die Funktion der Methode `net:into:` werden wir in dem nachfolgenden Abschnitt 5.2 genauer eingehen. Für jedes grafische Element, welches in einem Fenster verwendet wird, existiert eine zu `net:into:` äquivalente Methode. Desweiteren wird aus der Methode `net:into:` ein Objekt vom Typ `NetView` erzeugt, die wiederum ein Objekt vom Typ `NetController` erzeugt. Auf diese Art und Weise wird dann innerhalb des Builders des `ApplicationModels` die Struktur des Fensters aufgebaut, so daß schließlich die Oberfläche auf einem `GraphicsContext` dargestellt werden kann. In der Abbildung 3.4 haben wir die Benutzt-Beziehung zwischen einer Oberfläche und unserem Netz-IAT auf der Klassenebene dargestellt.

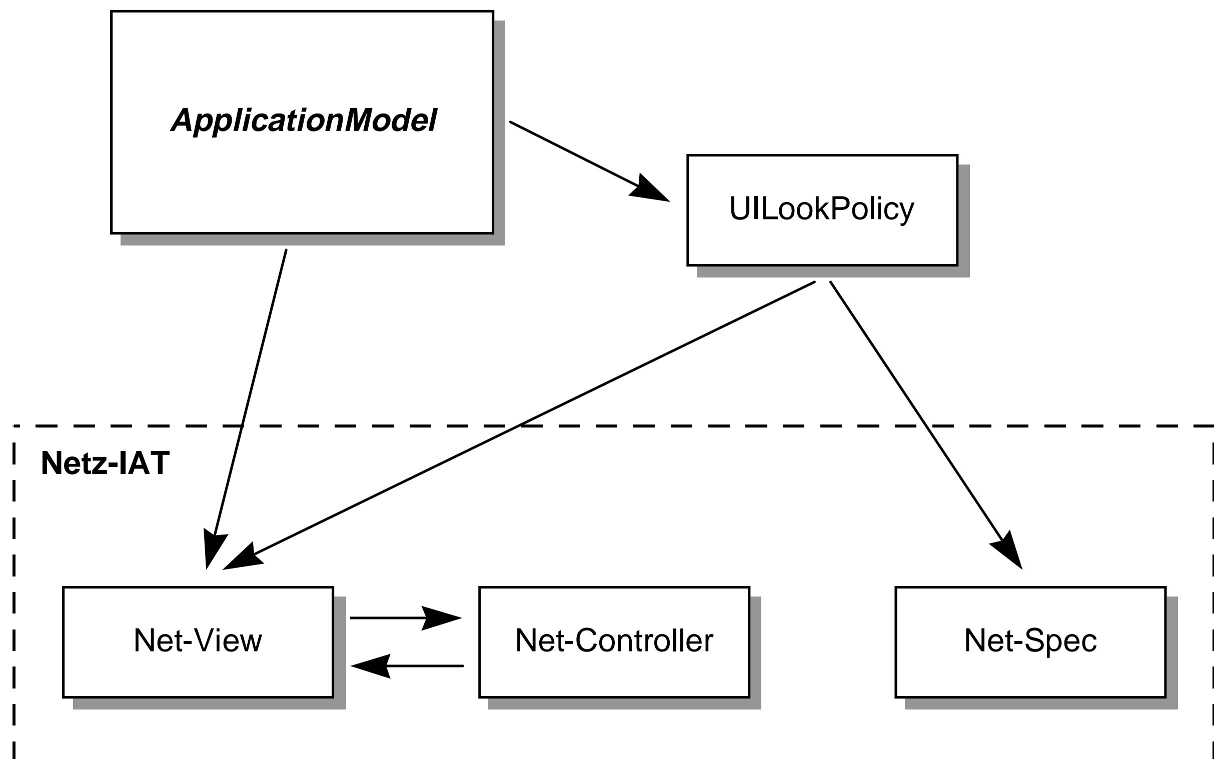


Abbildung 3.5

Zur Laufzeit der Anwendung kann man auf die Elemente der Oberfläche ausschließlich über die Variable `builder` des zugehörigen `ApplicationModels` zugreifen. Auf dieser Tatsache basiert unser Ansatz zur Bereitstellung einer Anwendungsschnittstelle für unseren Netz-Interaktionstypen. Variablen des Typs `UIBuilder` verstehen eine Methode mit dem Namen `componentAt: aSymbol`. Der Parameter `aSymbol` ist zum Zeitpunkt des Aufrufes die Identifikation des Widgets. Diese ID wird bereits während der Erstellung der Oberfläche mit dem Canvas-Tool über die Properties vom Anwender festgelegt. Innerhalb des MVC-Konzeptes ist eine View die zentrale Komponente, und somit zum Beispiel auch für die Erzeugung eines Controllers zuständig. Aus diesem Grunde haben wir uns dazu entschlossen, die Anwendungsschnittstelle unseres Netz-IAT in der `NetView` unterzubringen. Das jeweilige `ApplicationModel` kann sich dann über die folgende Befehlszeile die `NetView` besorgen:

```
myNetView := (builder componentAt: #MyIAT-ID) widget.
```

Die Methode `componentAt: aSymbol` liefert den `Wrapper` des eigentlichen Elementes zurück. Aus diesem Grunde muß man noch zusätzlich auf den zurückgelieferten `Wrapper` die Methode `widget` anwenden, um so direkt das Objekt vom Typ `NetView` des Netz-IAT zu bekommen. Über diese Art des Zugriffes auf den `UIBuilder` eines `ApplicationModels` kann man auf jedes beliebige in einem Fenster verwendete Objekt zurückgreifen. Das VisualWorks®-System bietet keine Möglichkeit, Zugriffe einzuschränken oder zu verhindern. Es ist also möglich, über diesen Einstieg auf die interne Struktur des Netz-IAT zuzugreifen. Abbildung 3.5 zeigt noch einmal das Prinzip der Benutzt-Beziehung zwischen dem `ApplicationModel` und der verwendeten `NetView`.

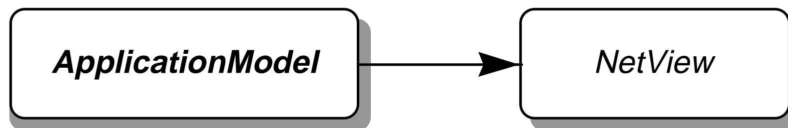


Abbildung 3.6

Bei der Verwendung dieses IAT's sind wir auf die Disziplin der Programmierer angewiesen, daß heißt bei der Verwendung sollte ausschließlich auf die `NetView` und die nachfolgend beschriebenen Methoden zurückgegriffen werden.

- `getNet`
  Die Anwendung dieser Methode auf ein `NetView`-Objekt liefert dem Aufrufer eine Kopie des zur Zeit bearbeiteten Netzes zurück. Der zurückgelieferte Wert ist vom Typ `Net`, das bedeutet, daß in ihm alle Informationen über die Knoten und Kanten des Netzes enthalten sind. Zu diesen Informationen gehören auch die Fensterkoordinaten, an denen sich ein Knoten befindet.

- `setNet:`
  Diese Methode erhält als Parameter ein Objekt vom Typ `Net` und führt dazu, daß das gerade aktuelle Netz des IAT's gegen dieses Netz ausgetauscht wird. Den Austausch des Netzes nehmen wir in der Form vor, daß ein neues Netz mit den Informationen aus dem als Parameter übergebenen Netz aufgebaut wird. Dies ist nötig, um eventuell vorhandene `Dependents` aus dem Netz-IAT herauszuhalten.

5 Einbettung des Interaktionstypen in VisualWorks®

Nachdem wir in den vorangegangenen Abschnitten die Struktur und Aufgabenverteilung in unserem Netz-IAT beschrieben haben, werden wir jetzt erläutern, wie dieser Interaktionstyp nun dem VisualWorks®-System zur Verfügung gestellt wird. Genauer gesagt handelt es sich hier um die Beschreibung der Vorgänge, die notwendig sind, um einen aus dem abstrakten Interaktionstypen abgeleiteten Netz-IAT in das VisualWorks®-System einzubetten. Allgemein ist hierzu zu sagen, daß für die Erweiterung des Canvas-Tool um eine neue Komponente drei Schritte notwendig sind.

1. Der erste Schritt besteht darin, die zu diesem Widget dazugehörigen Klassen - View, Controller, Model - zu implementieren.
2. Im Anschluß hieran ist eine Klasse zu implementieren, die die `componentSpec` beschreibt, in der das für alle VisualWorks®-Tools notwendige Klassen- und Instanzenprotokoll hinterlegt ist.
3. Im letzten Schritt schließlich wird die Klasse `UILookPolicy` um diejenige Methode erweitert, die es ermöglicht, das Widget aus den Spezifikationen heraus zu erzeugen. Desweiteren muß die Klasse `UIPalette` um das neu hinzugefügte Element erweitert werden.

Die Schritte 1 und 2 haben wir bereits in den Kapiteln 2 und 3 erläutert. Somit werden wir im folgenden den dritten Schritt beschreiben, wobei wir zuerst auf die Modifikation der Klasse `UIPalette` und dann auf die Erweiterung der Klasse `UILookPolicy` eingehen werden.

5.1 Registrierung des Interaktionstypen in der Palette

Um den Netz-IAT im Canvas-Tool verwenden zu können, müssen wir die Liste der Elemente erweitern, auf die das im Kapitel 2.4 kurz beschriebene Paletten-Fenster zurückgreift. Die Liste, in der die gerade aktiven Elemente verwaltet werden, heißt `ActiveSpecsList`. Um diese Liste zu erweitern, existieren innerhalb des VisualWorks®-Systems zwei Alternativen.

Ein Weg besteht darin, daß man die in der Klasse `UIPalette` existierende Methode `standardSpecsForPalette` benutzt, in der die zu verwendenden Elemente - List-Box, Edit-Feld, Label, usw. - mit dem Symbol des entsprechenden Klassennamens in einer `OrderedCollection` eingetragen sind. Der Klassenname bezieht sich in diesem Fall auf den jeweiligen Namen der `Specification`, in der die notwendigen Informationen hinterlegt sind und aus der schließlich auch die Methoden der Klasse `UILookPolicy` aufgerufen werden. Durch das Hinzufügen eines Symbols `#WidgetSpec` - `WidgetSpec` steht hier als Platzhalter für den Namen der von `NetSpec` abgeleiteten Klasse - zu dieser Liste wäre der Netz-IAT in der Palette verfügbar. Allerdings muß man in diesem Fall sicherstellen, daß die Methode `initialize` entweder an der Klasse `UIPalette` oder an dem Objekt selbst aufgerufen wird, da hieraus der Aufruf von `addSystemSpecs` bzw. `updateSystemSpecs` erfolgt, welche das Eintragen der Symbole in die `ActiveSpecsList` vornehmen.

Die zweite Variante der Erweiterung der ActiveSpecsList besteht darin, daß man die Liste über den Aufruf von

```
UIPalette activeSpecsList add: #WidgetSpec
```

direkt erweitert. Der Aufruf der Methode activeSpecsList an der Klasse UIPalette liefert die bereits angesprochene Liste ActiveSpecsList zurück, die mit den bekannten Methoden erweitert werden kann. In unserem Fall würde der Parameter #WidgetSpec durch den Namen der entsprechenden Spezifikationsklasse ersetzt werden. In dieser Variante werden alle weiteren Initialisierungen, die durch den Aufruf einer initialize-Methode hervorgerufen werden, vermieden.

Obwohl wir die erste Variante für den technisch besseren Weg halten, haben wir uns bei der Erweiterung der Palette an den zweiten Weg gehalten. Diese Entscheidung rührt daher, daß die Entwickler des VisualWorks®-System die zweite Methode für die Bereitstellung der Standardkomponenten vorgezogen haben. Im folgenden möchten wir noch auf eine Besonderheit hinweisen. Die oben erläuterte Codezeile ist in dem eigentlichen Quellcode nicht zu finden. Bearbeitet man allerdings die entsprechende Quellcode-Datei mit einem herkömmlichen Editor, so kann man diese Zeile zum Beispiel am Ende der Datei einfügen. Führt man den „File in“-Befehl auf dieser Datei aus, so interpretiert VisualWorks® diese Zeile als eine Anweisung, die sofort ausgeführt wird. Somit steht der neu definierte Netz-IAT dem Nutzer des Canvas-Tools sofort nach dem Einlesen der notwendigen Dateien zur Verfügung. Zu diesen Dateien gehören zum einen NetIAT.st, UILookPolicy-netinto.st und die Datei die die Spezialisierung der abstrakten Basisklassen enthält.

5.2 Funktion der Methode net:into: der Klasse UILookPolicy

Nachdem wir im Abschnitt 4.1 beschrieben haben, wie die Palette des Canvas-Tools von uns erweitert worden ist, möchten wir nun abschließend erklären, welche Aufgaben die Methode net:into: der Klasse UILookPolicy (Kategorie building) übernimmt.

Der Aufruf der Methode net:into: erfolgt aus einem Objekt der Klasse NetSpec heraus. Genauer gesagt erfolgt dieser Aufruf durch die in Abschnitt 3.3.6 beschriebene Methode dispatchTo: policy with: builder. Die Spezifikationsklasse NetSpec delegiert mit diesem Methodenaufruf die Erzeugung der Komponente an das entsprechende Policy-Objekt. Die Aufgabe der hier beschriebenen Methode besteht darin, aus einem Objekt vom Typ NetSpec das eigentliche Oberflächen-Element zusammen mit den dazugehörigen Models und Views zu erzeugen. Diese Methode wird immer dann aufgerufen, wenn es darum geht, einen Builder um ein Objekt vom Typ NetSpec zu erweitern. Da dies eine der wichtigsten Methoden im Hinblick auf die Darstellung des Netz-IAT's ist, werden wir im folgenden die einzelnen Abläufe sehr detailliert schildern.

In dieser Methode werden zwei lokale Variablen benutzt. Zum einen die Variable `component`, in der die dem `builder` hinzuzufügende Komponente - in diesem Fall ein Netz-IAT - hinterlegt ist. Zum anderen wird die Variable `mdl` verwendet, hinter der sich das für die Komponente benötigte Model verbirgt.

```
net: netSpec into: builder

| component mdl |
```

Zuerst besorgt sich die Methode das Model für diese `Specification`, die in der Klasse `NetSpec` hinterlegt ist, und auf die mit Hilfe der Methode `modelInBuilder:` zugegriffen werden kann. In diesem speziellen Fall wird ein Objekt vom Typ `Net` zurückgeliefert. In einem zweiten Schritt wird dann ein Objekt der Klasse `NetView` erzeugt, so daß diesem dann das Model zugewiesen werden kann.

```
  mdl := netSpec modelInBuilder: builder.
  component := NetView (netSpec defaultView) model: mdl.

  self setStyleOf: component to: netSpec style.
```

Mit der vorstehenden Codezeile wird dann dem in der Variablen `component` gespeicherten Objekt vom Typ `NetView` der für die Darstellung zu verwendende Font zugewiesen. Dieser Textstil ist ebenfalls in der `NetSpec` hinterlegt.

Für den Fall, daß man auf Nachrichten des Fenstersystems wie zum Beispiel Mausevents oder Fensternachrichten (`#close`, `#iconize`, usw.) reagieren möchte, muß ein sogenannter Dispatcher eingerichtet werden. Dieses Objekt wird in der Klasse `UIDispatcher` beschrieben und in der `Specification` eines grafischen Elementes festgelegt. Für unseren Netz-IAT benötigen wir keinen Dispatcher, da es sich hier um ein autarkes Element handelt, welches nicht vom Zustand eines oder mehrerer Fenster abhängig ist. Unabhängig hiervon ist es aber notwendig, diesen Methodenaufruf durchzuführen, da ansonsten der `Application-Controller` der Anwendung in seiner `ControlLoop` für unseren Netz-IAT keinen gültigen Wert vorfindet und somit einen Programmabbruch forciert.

```
  self
      setDispatcherOf: component
      fromSpec: netSpec
      builder: builder.
```

In den nächsten Schritten müssen die Eigenschaften des Netz-IAT's festgelegt werden. Zuerst wird in diesem Zusammenhang festgelegt, ob der Netz-IAT den Fokus der Anwendung bekommen kann. Dieses wird über die Methode `initallyEnabled` bei der `NetSpec` abgefragt und dann in der Variablen `state` der `NetView` abgelegt. Hinter der Variablen `state` verbirgt sich ein Objekt vom Typ `WidgetState`.

```
component widgetState isEnabled: netSpec initiallyEnabled.

builder isEditing
    ifFalse:
    [component widgetState isVisible: netSpec
    initiallyVisible.
    component setNetRepresentation:
        ((Smalltalk at: (netSpec netRepresentation)) new).
    component setNetGrammar:
        ((Smalltalk at: (netSpec netGrammar)) new)].
```

Solange mit Hilfe des Canvas-Tool eine Oberfläche entworfen wird, liefert der Aufruf der Methode `isEditing` an dem `builder` den Wert `true`. Im Gegensatz hierzu liefert `isEditing` zum Zeitpunkt des Starts einer Anwendung den Wert `false`. Solange der Wert `true` zurückgeliefert wird, dürfen bestimmte Parameter für den Netz-IAT noch nicht gesetzt werden. Zu diesen gehört der Eintrag, ob der IAT beim Start der Oberfläche sichtbar ist, das Eintragen eines Objektes, das die Darstellungsform für das Netz festlegt und die Festlegung der dem Netz zugrunde liegenden Grammatik. Über die globale Variable `Smalltalk` kann man mit der Methode `at:` auf alle Namen der in dem VisualWorks®-System vorhandenen Klassen zugreifen. Mit Hilfe der Methoden `netRepresentation` und `netGrammar` bekommt man die in den Properties eingetragenen Klassennamen als Strings zurück. Mit diesen Werten als Schlüssel suchen wir dann in dem `SystemDictionary Smalltalk` nach den Klassennamen. Den dann zurückgelieferten Klassennamen verwenden wir, um ein Objekt des entsprechenden Typs zu erzeugen.

Nachdem wir unseren Netz-IAT vollständig initialisiert haben, tragen wir diesen nun mit der Methode `component:` in den Builder des `ApplicationModels` ein.

```
builder component: component.
```

Unser Netz-IAT soll einen Rahmen besitzen und die Möglichkeit haben, horizontale und vertikale Scrollbalken anzuzeigen. Diese Aufgaben werden durch einen Wrapper erledigt, der mit der Methode `manufactureGeneralWrapperFor:into:` erzeugt wird. Desweiteren wird bei diesem Methodenaufruf der so erzeugte `Wrapper` in der Variablen `container` der `NetView` eingetragen.

```
self manufactureGeneralWrapperFor: netSpec into: builder.
```

Scrollbalken reagieren in diesem System auf ein Mausklick normalerweise mit dem scrollen um ein Pixel in die gewünschte Richtung. Für die Anzahl der pro Mausklick zu scrollenden Pixel ist ein in dem Wrapper der `NetView` enthaltener Offset zuständig. Auf diesen greifen wir mit der Methode `scrollOffsetHolder` zu und setzen das Raster auf einen Wert von 10 Pixeln in horizontaler und vertikaler Richtung.

```
component container scrollOffsetHolder grid: 10 @ 10.
```

In dem nächsten Schritt erfolgt die Zuweisung eines Layouts zu dem `Widget`. Hinter diesem Layout verbirgt sich ein Objekt vom Typ `Rectangle`, das die Ausmaße des Netz-IAT's festlegt. Bei der Benutzung eines Netz-IAT's innerhalb des Canvas-Tools werden hierfür Standard-Werte benutzt.

```
builder applyLayout: netSpec layout.
```

Da jedes Widget in VisualWorks® von einem `Wrapper`, deren Aufgaben wir in Abschnitt 2.2.1.4 erläutert haben, umgeben ist, müssen wir auch für unseren Netz-IAT einen `Wrapper` erzeugen. Dieses geschieht über die Methode `simpleWidgetWrapperOn:spec:-state:`, der einen `SpecWrapper` erzeugt.

```
builder wrapWith: (self
        simpleWidgetWrapperOn: builder
        spec: netSpec
        state: builder component widgetState)
```

6 Ein Anwendungsbeispiel

Das weitergehende Ziel dieser Arbeit basiert auf einem Ausschnitt der Promotion von Guido Gryzcan, in dem die Vergegenständlichung kooperativer Arbeit durch Prozeßmuster behandelt wird ([6]). Wir fassen in dieser Arbeit ein Prozeßmuster als die Spezialisierung eines allgemeinen Netzes auf, so daß der hier vorgestellte IAT in der Lage ist, Prozeßmuster darzustellen und zu editieren. Somit besteht also die Möglichkeit, Prozeßmuster in Form eines Netzes zu versenden, die dann auf einem anderen Arbeitsplatz innerhalb des dortigen Netz-IAT's weiterbearbeitet werden können.

6.1 Ein Netz-IAT zur Bearbeitung von Prozeßmustern

Unser Ziel besteht darin, auf Basis des von uns implementierten Netz-IAT einen Interaktionstypen durch Spezialisierung einiger Klassen zu entwerfen. Die Spezialisierung betrifft folgende Klassen:

- `NodeView`
- `EdgeView`
- `NetRepresentation`
- `NetGrammar`

Im vorwege haben wir bereits erwähnt, daß wir ein Prozeßmuster als die Spezialisierung eines allgemeinen Netzes betrachten wollen. Bei der Betrachtung eines Prozeßmusters stellen wir jedoch fest, daß man die Knoten eines Prozeßmusters als erledigt markieren kann. Dieser Zustand wird zum Beispiel durch einen Haken in dem Prozeß dargestellt. Diese Tatsache bedeutet, daß ein Knoten in einem Prozeßnetz mehr Zustände besitzt, als ein allgemeines Netz. Desweiteren soll es bei diesem Interaktionstypen möglich sein, aus der den IAT benutzenden Anwendung heraus, Bezeichnungen für die zur Verfügung stehenden Prozesse zu setzen. Für uns bedeutet dieses, daß wir auch die Applikationsschnittstelle des Netz-IAT erweitern müssen. Insgesamt folgt hieraus, daß wir nicht nur die bereits angeführten Klassen ändern, sondern auch in die Modellierung unseres Netzes eingreifen müssen. Insgesamt umfaßt diese Änderung folgende Klassen:

- `GraphicalNode`
- `NetSpec`
- `NetView`
- `NetController`

Wir werden nun, angefangen mit den zuletzt genannten Klassen, die Änderungen beschreiben, die notwendig sind, um den Interaktionstypen auf Prozeßmuster anzupassen

- `GraphicalNode`
  Zunächst erzeugen wir eine Klasse `GraphicalProcessNode`, die von der Klasse `GraphicalNode` erbt. Diese neue Klasse erweitern wir um die Exemplarvariable `checked`, die anzeigt, ob ein Haken an dem Prozeß gesetzt ist oder nicht. Da wir der Klasse eine neue Variable hinzugefügt haben, ist es notwendig, die `initialize`-Methode zu redefinieren, um eine korrekte Initialisierung der Variablen `checked` sicherzustellen. Abschließend müssen wir in der Kategorie `accessing` noch die Methoden `checked` und `toggleMark` implementieren, die den Zugriff auf diese Variable realisieren.

- `NetSpec`
  Ein Objekt vom Typ `NetSpec` liefert nach der Anwendung der Methode `defaultView` ein Objekt der View-Klasse zurück, die für diesen IAT zu verwenden ist. Wie wir bereits beschrieben haben, wird die Applikationsschnittstelle des Netz-IAT in der Klasse `NetView` definiert. Da diese Schnittstelle nun erweitert werden muß, erzeugen wir die Klasse `ProcessNetSpec` als Unterklasse von `NetSpec`. Dort redefinieren wir dann die Methode `defaultView` in der Form, daß sie ein Objekt der Klasse `ProcessNetView` zurückliefert.

- `NetView`
  Innerhalb der Klasse `NetView` müssen von uns der Benachrichtigungsmechanismus und die Applikationsschnittstelle angepaßt werden.. Hierfür erzeugen wir eine Klasse `ProcessNetView`, die von der `NetView` erbt. Sowie die Bezeichnung eines in dem Netz enthaltenen Prozesses verändert wurde, wird die `ProcessNetView` darauf mit der Nachricht `#NewProcessLabel` hingewiesen. Da diese Nachricht in dieser Form noch nicht in der Oberklasse vorhanden ist, müssen wir nun innerhalb der Methode `upcastEvent:with:from:path:` der Klasse `ProcessNetView` die Verarbeitung dieser Nachricht implementieren. Alle anderen Nachrichten werden an die gleichnamige Methode der Oberklasse weitergeleitet. Die Änderung der Applikationsschnittstelle zieht in diesem Fall die Änderung der Klasse `NetController` nach sich, da das Kontextmenü veränderbar sein soll, welches über ein Objekt vom Typ `NetController` erzeugt wird. Innerhalb der Klasse `ProcessNetView` redefinieren wir aus diesem Grund die Methode `defaultControllerClass` in der Form, daß sie anstatt des Klassennamens `NetController` `ProcessNetController` zurückliefert. Auf Anforderung durch die den IAT verwendende Anwendung, muß eine tiefe Kopie des gerade bearbeiteten Netzes zurückgeliefert werden. Hierfür wird die Methode `copyNet` verwendet. In dieser Methode müssen wir durch den Austausch der verwendeten Knotenklassen sicherstellen, das nur Objekte vom Typ `GraphicalProcessNode` verwendet werden. Zusätzlich wird von uns das Protokoll der Applikationsschnittstelle um die beiden nachfolgend beschriebenen Methoden erweitert.

- `setProcessLabelList:`
  Diese Methode ermöglicht es, für verschiedene Prozeßtypen Bezeichnungen in der Auswahlliste des Kontextmenüs zu hinterlegen. Der zu übergebene Parameter ist eine `OrderedCollection`, deren Elemente vom Typ String sind und die Bezeichnung des Prozesses darstellen.
- `getProcessLabelList`
  Die Anwendung dieser Methode liefert die zur Zeit in dem Prozess-IAT verfügbaren Bezeichnungen in einer OrderedCollection zurück.

- `NetController`
  Die Notwendigkeit einer Spezialisierung der Klasse `NetController` rührt von der Erweiterung bzw. geänderten Handhabung der Menüeinträge und der Tatsache, daß wir eine neue Klasse `GraphicalProcessNode` eingeführt haben. Letzteres bedeutet, daß wir in den Methoden `newNode:` und `newNode:withLabel:` der neuen Klasse `ProcessNetController` die Klasse `GraphicalNode` gegen `Graphical-ProcessNode` austauschen müssen. Die Veränderungen hinsichtlich des Kontextmenüs betreffen die Methoden `menu`, `setMenu` und `toggleMark`. Die beiden zuerst genannten Methoden sind für den Aufbau des Menüs zuständig. Wir redefinieren diese Methoden, in dem wir zuerst die jeweilige Methode der Oberklasse aufrufen und das resultierende Menu dann mit unseren Erweiterungen versehen. Die Methode `toggleMark` benötigen wir, um aus dem Menü heraus den Befehl geben zu können, einen Prozeß mit einem Haken zu versehen.

Es sei an dieser Stelle betont, daß die beschriebenen Änderungen aufgrund der Tatsachen, daß die Schnittstelle des IAT's zu einer Anwendung erweitert werden mußte und ein Prozeß mehr Zustände besitzt als ein Knoten eines allgemeinen Netzes, notwendig waren. Die nun folgende Erläuterung der Änderungen an den verschiedenen Klassen sehen wir als den Standardfall für Anpassungen unseres Netz-IAT an.

- `NodeView`
  Die Klasse `NodeView` repräsentiert eine abstrakte Oberklasse, von der alle Klassen erben, die Knoten eines Netzes darstellen sollen. Für den Prozeßmuster-IAT verwenden wir eine Klasse mit dem Namen `ProcessNodeView`, die von der Klasse `NodeView` erbt. Erweiterungen der Schnittstelle sind hier nicht notwendig, da die zu implementierenden Methoden bereits von der Klasse `NodeView` vorgegeben werden. Wir müssen also die Methoden `displayNormalOn:` und `displaySelectedOn:` implementieren, die die Darstellung eines Prozesses im normalen oder selektierten Zustand übernehmen. In diesen Methoden verwenden wir die Hilfsmethode `displayMark:withBounds:` zur Darstellung eines Hakens in einem Prozeß. Im weiteren müssen wir die Methode `preferredBounds` implementieren, die angibt, wie groß die benötigte Fläche ist, um den Prozeß mit seiner Bezeichnung darstellen zu können. Die `ProcessNodeView` wird über Änderungen an ihrem Model über Nachrichten informiert. In der Methode `update:` entscheidet die View nun, ob sie lokal diese Nachrichten verarbeiten muß, oder ob diese Nachrichten an die `ProcessNetView` weitergeleitet werden müssen. Die Redefinition dieser Methode wurde notwendig, da das zugrunde liegende Model, die Klasse `GraphicalProcessNode`, mehr Nachrichten versendet als die ursprüngliche Klasse `GraphicalNode`.

- `EdgeView`
  Die Änderungen an der Klasse `EdgeView` gleichen zum großen Teile denen an der Klasse `NodeView`. In der von uns eingeführten Klasse `ProcessEdgeView` sind die Änderungen an der Darstellungsweise einer Kante (Pfeildarstellung) in die Methoden `displayNormalOn:` und `displaySelectedOn:` eingebracht worden. Die Änderung der `update:`-Methode entfällt hierbei, da keine neue Nachrichten im Bezug auf Kanten verarbeitet werden müssen.

- `NetRepresentation`
  Äquivalent zur Klasse `NodeView` ist auch die Klasse `NetRepresentation` eine abstrakte Klasse. In dieser Klasse legen wir die für die Darstellung eines Prozeßmusters zu verwendenden View-Klassen fest. In der für den Prozeßmuster-IAT gedachten Klasse `ProcessRepresentation` implementieren wir nur die von der Oberklasse vorgegebenen Methoden. Die Funktionalität entspricht dem in Abschnitt 4.3.3 beschriebenem Umfang.

- `NetGrammar`
  Dem Prozeßmuster-IAT liegt eine Grammatik vom Typ `ProcessGrammar` zugrunde. In dieser definieren wir wie von der Oberklasse vorgegeben die gültigen Typen für Knoten und Kanten. Für nähere Informationen verweisen wir auf Kapitel 4.3.2.

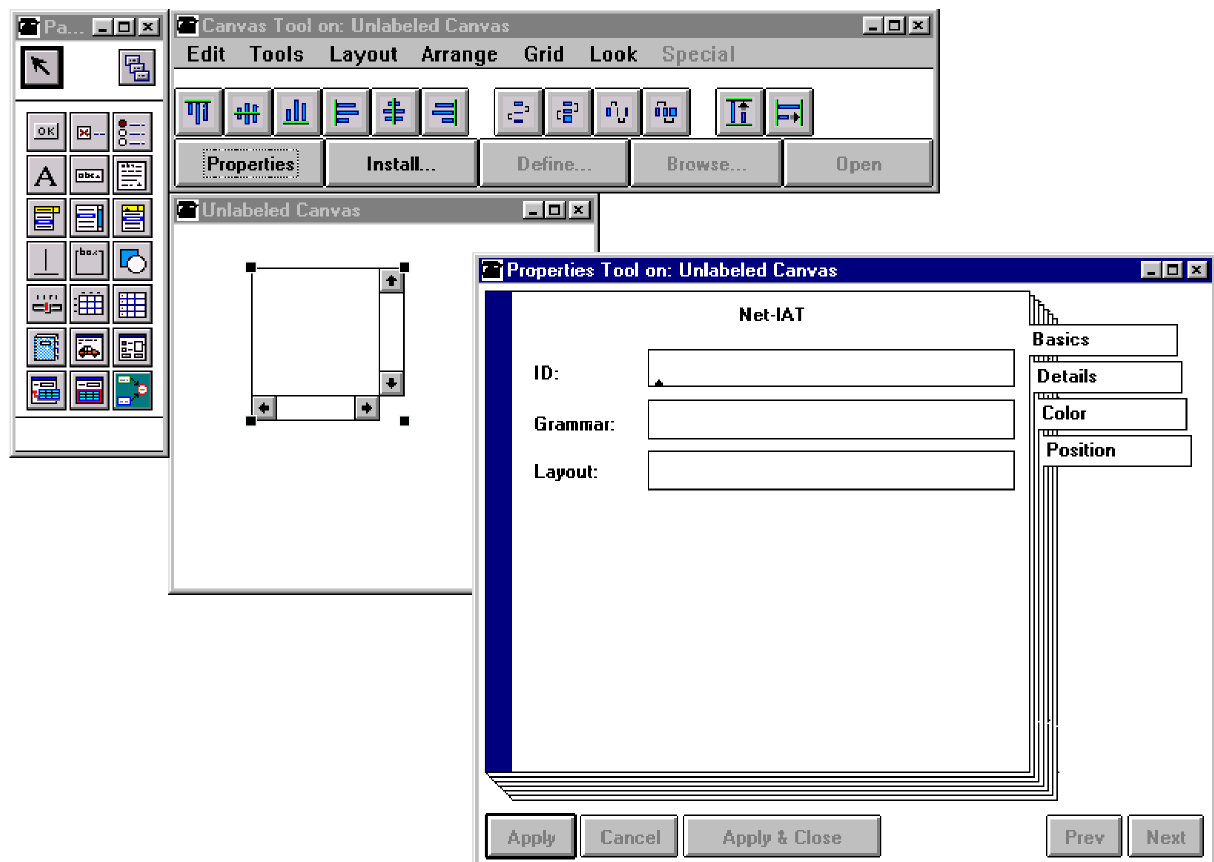


Abbildung 5.1

Mit den bis hier beschriebenen Änderungen ist die reine Erstellung eines Interaktionstypen zur Bearbeitung von Prozeßmustern abgeschlossen. Wir können im Moment aber in dem Canvas-Tool noch nicht auf einen Prozeßmuster-IAT zugreifen. Deswegen müssen wir an dieser Stelle das Palettenfenster um dieses von uns neu erstellte Widget erweitern. Die hierfür notwendigen Schritte haben wir im Kapitel 5.1 bereits beschrieben. Die Methode `net:into:` der Klasse `UILookPolicy` müssen wir nicht ändern bzw. reimplementieren, um den Prozeßmuster-IAT dem System zur Verfügung zu stellen. Solange man bei der Erweiterung bzw. Spezialisierung des Netz-IAT's darauf achtet, daß die Klasse `NetSpec` nicht angepaßt werden muß, ist es nicht notwendig das Palettenfenster zu erweitern.

Wir möchten nun im Anschluß zeigen, wie man diesen neuen IAT nun in die Oberfläche einer Anwendung integrieren kann.

Nachdem man das Canvas-Tool gestartet hat und einen Prozeßmuster-IAT aus der Paletten-Leiste in das Arbeitsfenster gezogen hat, kann man die Größe und Position dieses Interaktionstypen festlegen. Danach ruft man über den Properties-Button dasjenige Fenster auf, welches es ermöglicht die Eigenschaften dieses IAT zu editieren. Abbildung 5.1 zeigt exemplarisch das resultierende Fenster. In dem mit „Basics" bezeichneten Abschnitt des Properties-Tools wird nun eine eindeutige ID für den IAT eingetragen, mit deren Hilfe die Anwendung zur Laufzeit auf die Schnittstelle des Interaktionstypen zugreifen kann. Außerdem werden in den Feldern Grammar und Layout die Namen derjenigen Klassen eingetragen, auf die der IAT für die syntaktische Überprüfung bzw. für die Darstellung des Netzes zurückgreifen soll.

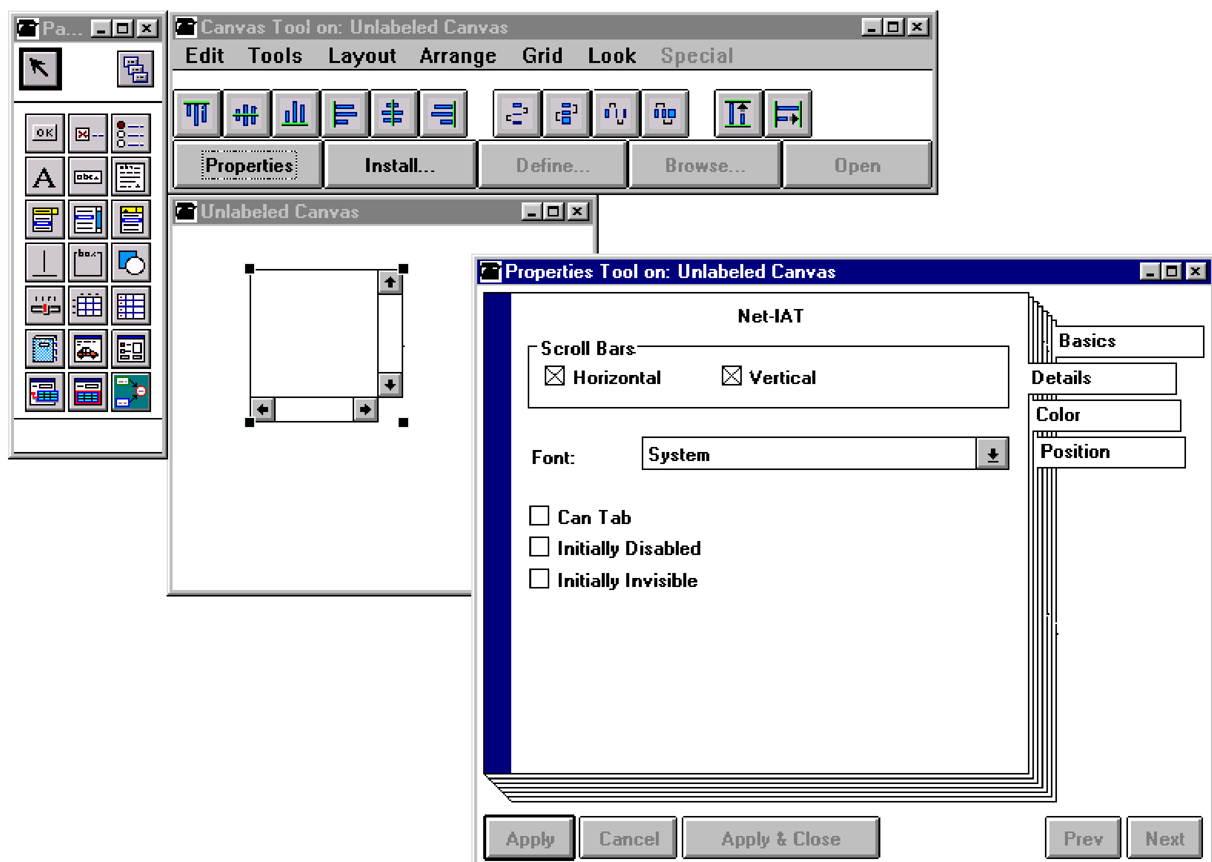


Abbildung 5.2

Zusätzlich zu der Kategorie „Basics“ existieren in dem Properties-Fenster noch die Kategorien „Details“, „Color“ und „Position“. Diese drei Funktionen beschäftigen sich mit der weiteren Darstellung dieses Prozeßmuster-IAT’s zur Laufzeit der Anwendung. Wir werden hier allerdings nur noch kurz auf die Eigenschaften eingehen, die sich hinter dem Begriff „Details“ verbergen. Abbildung 5.2 zeigt das Details-Fenster mit den dort editierbaren Eigenschaften des Prozeßmuster-IAT’s. Zu diesen gehört zum Beispiel die Angabe, ob horizontale und / oder vertikale Scrollbalken verwendet werden sollen. Desweiteren kann hier der zu verwendende Font eingestellt werden. Zusätzlich wird in diesem Fenster festgelegt, ob der IAT über die Tabulator-Taste erreichbar ist („Can Tab“), ob der Interaktionstyp zum Zeitpunkt des Starts der Anwendung inaktiv sein soll („Initially Disabled“) und ob der Prozeßmuster-IAT beim Start der Anwendung nicht auf der Oberfläche erscheinen soll („Initially Invisible“).

Nachdem man nun alle diese Eigenschaften in dem Properties-Tool versorgt hat, kann man mit Hilfe des Install-Buttons ein `literalArray`, welches eine `FullSpec` des Fensters enthält, in eine vom Benutzer vorgegebene Klasse schreiben. Man findet dieses Array in der entsprechenden Klassensicht unter der Kategorie `interface specs`. Der Define-Button hat bezogen auf unseren IAT keine Funktion, da wir keinen `AspectAdaptor` für die Kopplung des Oberflächenelementes mit seiner Anwendung verwenden.

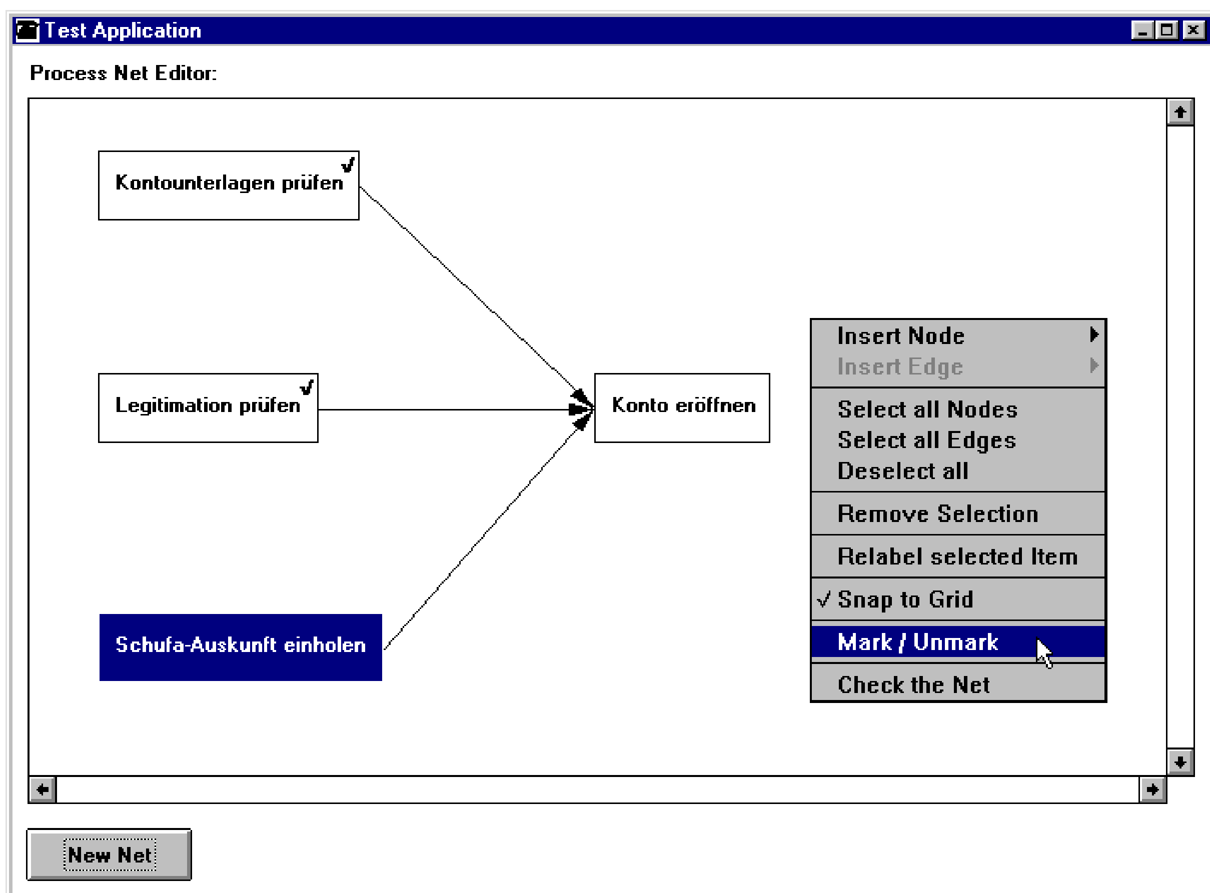


Abbildung 5.3

Nach dem Start der so erzeugten Oberfläche ist der Anwender in der Lage, Prozeßmuster zu erstellen bzw. dem IAT übergebene Prozeßmuster zu editieren. Abbildung 5.3 zeigt beispielhaft die Erstellung eines Netzes mit unserem neu erstellten Interaktionstypen.

7 Ausblick

Die Darstellungsmittel der Netztheorie werden aufgrund ihrer bildhaften und deswegen auch leicht verständlichen Form sehr häufig innerhalb der Informatik verwendet, um komplexe Zusammenhänge anschaulich darzustellen.

> „Netztheorie wird eingesetzt wegen ihrer Mächtigkeit und wegen ihrer guten Anpassung an das Vermögen des Menschen, bildliche Strukturen und Vorgänge zu erfassen und zu verarbeiten. Insbesondere bietet sie Vorteile bei der Darstellung nebenläufiger Handlungen und Funktionseinheiten und deren Strukturierung durch Vergröberung über mehrere Abstraktionsschichten hinweg.“ ([7], Einleitung)

Aufgrund der sich hieraus ergebenden Vorteile ist es wünschenswert, in Softwareapplikationen auf ein Oberflächenelement zurückgreifen zu können, mit dem man Netze in jeglicher Form bearbeiten kann. Im Hinblick auf die am Arbeitsbereich Softwaretechnik vertretene Lehre bedeutet dies, daß ein Interaktionstyp für Netze eingeführt wird, der in der Lage ist, ein Netz darzustellen und Möglichkeiten zur Manipulation des Netzes bietet. Dieses haben wir versucht mit der vorliegenden Arbeit zu realisieren. Zum Abschluß dieser Arbeit möchten wir aufzeigen, wo aus unserer Sicht die Stärken und Schwächen der von uns gewählten Implementierung liegen und im weiteren auf mögliche Erweiterung dieses Interaktionstypen hinweisen.

7.1 Diskussion der Implementierung

Wir werden im folgenden die Diskussion der von uns gewählten Implementierung in Anlehnung an den von uns in den Kapiteln 4.1 und 4.2 aufgestellten Zielen vornehmen. Hierbei werden wir feststellen, inwiefern wir die von uns gesetzten Ziele erreicht haben.

Der Netz-Interaktionstyp steht uns bei der Gestaltung von grafischen Benutzeroberflächen mit Hilfe des Canvas-Tools zur Verfügung. Um dieses Ziel zu erfüllen, haben wir das VisualWorks®-System um die notwendigen Klassen und Methoden in der Form erweitert, daß man durch das Laden der entsprechenden Dateien Zugriff auf diese neue Komponente bekommt.

Ein weiteres Ziel, welches wir erreichen wollten, bestand darin, mit Hilfe eines Interaktionstypen für Netze jedweder Anwendungssoftware eine definierte Schnittstelle zur Verfügung zu stellen und somit die Funktionalität des Fenstersystems gegenüber der Anwendung zu kapseln. Dieses ist uns in soweit gelungen, als das wir in der Klasse `NetView` eine Applikationsschnittstelle implementiert haben. Aufgrund des Aufbaus von grafischen Benutzeroberflächen in dem VisualWorks®-System ist es uns aber nicht möglich, den Zugriff der Anwendung auf bestimmte Methoden zu beschränken. Somit ist es zum jetzigen Zeitpunkt durchaus möglich, daß sich Anwendungen über den `builder` Zugang zu der internen Struktur des Interaktionstypen verschaffen.

Wenn wir voraussetzen, daß sich die Entwickler bei der Verwendung unseres Netz-IAT's an die vorgegebene Schnittstelle halten, so ist es unserer Meinung nach berechtigt, diesen Interaktionstyp funktionale Eigenschaften zuzusprechen. Wir sind in der Lage, dem Interaktionstypen als Parameter einen fachlichen Wert Netz zu übergeben, diesen Wert dann mit Hilfe des Netz-IAT zu bearbeiten und liefern eine tiefe Kopie des Netzes auf Anforderung an die Anwendung zurück. Zudem kann bei der Benutzung der Applikationsschnittstelle nicht auf die internen Zustände des IAT zugegriffen werden.

Ursprünglich war es außerdem unser Ziel, eine Vermischung zwischen der grafischen Information (Koordinaten der Knoten) und den Bestandteilen eines Netzes zu vermeiden. Da wir aber bei der Modellierung des fachlichen Wertes Netz auf die in dem VisualWorks®-System vorhandenen `Set`-Klassen zurückgegriffen haben, konnten wir dieses Ziel nicht erfüllen. Um innerhalb des Netzes die Eindeutigkeit von Knoten gewährleisten zu können, haben wir `Sets` verwendet, da diese bereits die gewünschte Funktionalität bereitstellen. Diese Entscheidung hat aber für uns den Nachteil, daß wir keinen Einfluß darauf haben, an welcher Position innerhalb eines `Sets` ein Knoten bzw. eine Kante eingefügt wird. Dieses ist aber im Bezug auf die Abtrennung der grafischen Information notwendig, um eine eindeutige Zuordnung der jeweiligen Koordinaten zu den einzelnen Knoten vornehmen zu können. Letztendlich haben wir uns dazu entschlossen in dem unserem Netz-IAT zugrunde liegenden Model Knoten vom Typ `GraphicalNode` zu verwenden. Somit liefern wir über die Methode `getNet` der Applikationsschnittstelle ein Netz, in dem sowohl die Netzbestandteile als auch die Koordinaten der Knoten enthalten sind. Dieses Manko ließe sich dadurch beheben, daß man zusätzlich eine Relation zwischen den Knoten und einer separat verwalteten grafischen Information vorhält.

Insgesamt besehen ist unser IAT allerdings relativ einfach an verschiedene Darstellungsweisen und Grammatiken anzupassen. Sofern man allerdings neue Zustände an den Elementen des Netzes benötigt, ergibt sich die von uns im Kapitel 6.1 beschriebene Situation (Prozeßmuster-IAT). In diesem Fall müssen Änderungen bis hinunter auf die Ebene der Spezifiaktionsklasse vorgenommen werden. Unserer Meinung nach zeigt das Kapitel 6 aber auch, daß sich auf Basis des Netz-IAT mit vertretbarem Aufwand ein neuer Interaktionstyp entwickeln lässt.

Die sich aus der vorangegangenen Darstellung der nicht oder nur teilweise erreichten Ziele ergebenen weitergehenden Aufgaben haben wir dem folgenden Abschnitt 7.2 zugrunde gelegt.

7.2 Mögliche Erweiterungen

Das vornehmliche Ziel der Erweiterung dieses Interaktionstypen ist in unseren Augen, die Entwicklung von Konzepten für die Implementierung von verschiedenen Grammatiken, die es auf einfache und effektive Art und Weise zulassen auch kompliziertere Netztypen zu beschreiben und zu bearbeiten. Beispiele hierfür sind zum Beispiel die Klasse der kontextfreien Grammatiken (CFG), wobei man hierbei noch untersuchen muß, in wie weit kontextfreie Grammatiken zur Beschreibung von Netzen geeignet sind.

Im weiteren ist es sinnvoll, Mechanismen einzubauen, die es verhindern, über den Builder eines `ApplicationModels` auf alle Bestandteile des IAT zugreifen zu können. Dieses würde den Vorteil haben, daß man über die Anwendungsschnittstelle des Netz-IAT zum Beispiel nicht mehr auf Methoden der NetView zugreifen kann. In diesem Zusammenhang ist es ebenfalls denkbar, den Interaktionstypen vermehrt auf das Werkzeug - Automat - Material Konzept auszurichten. Inwieweit dieses bezogen auf das von uns eingesetzte VisualWorks®-System machbar ist, kann von uns zum jetzigen Zeitpunkt nicht abgeschätzt werden.

Im Hinblick auf die vielfältigen Einsatzbereiche von Netzen im allgemeinen, ist es außerdem sinnvoll, diesen IAT auf eine andere Programmiersprache zu portieren, um auf mehreren Systemen die Möglichkeit zu haben, auf einen Netz-IAT zurückgreifen zu können. Unserer Meinung nach bietet sich hierfür die Sprache Java an.

8 Literaturverzeichnis

[1] C. Bücker, J. Geidel, M. F. Lachmann: *Programmieren in Smalltalk mit VisualWorks®: Smalltalk nicht nur für Anfänger - 2. erw. Auflage.* Springer, 1995.

[2] Howard: *The Smalltalk Developer's Guide to VisualWorks.* SIGS Books, 1995.

[3] ParcPlace®: *VisualWorks® Object Reference.* ParcPlaceSystems, 1994.

[4] ParcPlace®: *VisualWorks® Cookbook.* ParcPlaceSystems, 1994.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley Verlag, 1996.

[6] M. Wulff, G. Gryczan, H. Zuellighoven: *Prozessmuster für die situierte Kooperation kooperativer Arbeit.* In: H. Kremar, H. Lewe, G. Schwabe (Hrsg.): Herausforderung Telekooperation, D-CSCW'96, Springer 1996, Seite 89 - 103.

[7] E. Jessen, R. Valk: *Rechensysteme: Grundlagen der Modellbildung.* Springer, 1987.

[8] H. Kilberth, G. Gryczan, H. Zuellighoven: *Objektorientierte Anwendungsentwicklung - Konzepte, Strategien, Erfahrungen.* Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

9 Anhang (Quellcode)

Der zu dieser Arbeit gehörende Quellcode, so wie eine Beispielapplikation stehen auf einer Web-Seite mit der folgenden Adresse zur Verfügung:

http://swt-www.informatik.uni-hamburg.de/~3lammers/Studienarbeit/