Bedienhandbuch PnEd

Markus Rother (8544832)

January 11, 2015

In diesem Bedienhandbuch wird der *PnEd* - ein einfacher Petri-Netz Editor vorgestellt. Dieser, sowie die hiermit vorliegende Dokumentation sind als Aufgabenbearbeitung des Grundpraktikum Programmierung (Kurs 01584) im Wintersemester 2014/15 an der Fernuniversität in Hagen entstanden. Der Editor wurde von mir (Markus Rother) in Java implementiert, und mit dem Namen *PnEd* versehen.

Diese Dokumentation skizziert meine wichtigsten Designentscheidungen und wie sie entstanden sind. Anschließend wird die Grundlegende Architektur der Software vorgestellt, also die Strukturierung der `packages` und der wichtigsten Klassen. Die Bedienanleitung beschreibt detailliert die Verwendung des Editors. Abschließend wird das größte bekannte Problem kurz beschrieben.

Contents

1 Design des PnEd

1.1 Loose Kopplung und Single-Responsibility-Prinzip

Zu Beginn der Bearbeitungsphase stürzte ich mich kopfüber in die Implementierung eines Datenmodells. Als ich dann begann, an der Benutzeroberfläche zu arbeiten, stellte sich rasch heraus, dass die beiden nicht so recht zusammenpassen wollten. Ich musste das Datenmodell häufig an die Anforderungen der Benutzeroberfläche anpassen. Außerdem war vieles am Datenmodell unnötig.

Es bestand ganz offensichtlich eine sehr starke Kopplung zwischen meinem Datenmodell und der Benutzeroberfläche, die mir missfiel. Ich entschied mich, das Datenmodell vollständig zu ignorieren und zunächst die Oberfläche zu schreiben. Ich wollte bei der Erstellung der Oberfläche so wenig Annahmen wie möglich über das zugrunde liegende Datenmodell machen.

Dazu sollte die Verbindung zwischen Datenmodell und der Benutzeroberfläche über Ereignisse (events) stattfinden. Ähnlich einer Client-Server Architektur sollte die Benutzeroberfläche mit dem Datenmodell mittels Nachrichten sprechen. Ich bräuchte dann lediglich einen Verbindungsstrang, über den diese Nachrichten, Ereignisse oder Anfragen kommuniziert würden.

Die Benutzeroberfläche selbst sollte dabei frei von jeglicher Logik bleiben, zum Beispiel nicht für das Berechnen des Netzes bei Schaltung von Transitionen verantwortlich sein. Und das Datenmodell sollte keinerlei Kenntnis des Zustands der Visualisierung haben, nicht einmal, welche Knoten gerade ausgewählt wären.

Mit diese puristischen herangehensweise gelang es mir zu meiner eigenen Überraschung, die Benutzeroberfläche fast vollständig - sozusagen als eine Hülle - zu implementieren, ohne mich im geringsten um die Logik des Petri-Netzes zu kümmern. Zu meiner noch größeren Überraschung war das Anschließende implementieren des Datenmodells eine Arbeit von kaum zwei Stunden. Noch einfacher war die Anbindung des vorgegebenen PNMLParsers, da auch dieser wie die Benutzeroberfläche lediglich Nachrichten verschicken musste, also etwa für das Erstellen eines Knotens eine Nachricht mit Typ, Namen und Koordinaten. Diese Nachricht würde dann sowohl vom Datenmodell als auch von der Benutzeroberfläche entsprechend interpretiert.

Kritisch resümierend kann ich folgendes festellen: Auch wenn ich mir im Verlauf des Projekts durch mein Vorgehen zahlreiche andere Probleme einhandelte, halte ich meine Entscheidung für gut. Ich habe erfahren, wo die Grenzen einer solchen Trennung liegen, ich konnte mich mit Nebenläufigkeit-

und musste mich mit einigen Details des `swing` frameworks befassen. Es wäre sicherlich vieles einfacher gewesen, aber bei weitem nicht so erkenntnisreich.

1.2 Aufbau der Pakete

Das Projekt ist in 23 Pakete unterteilt. Diese können weiter in drei größere Bereiche gruppiert werden:

1. Pakete, deren Klassen die Kernfunktionalität des *PnEd* auszeichnen. Ohne diese Klassen funktioniert nichts, sie funktionieren aber ohne andere Klassen des Editors.
2. Pakete, deren Klassen die Funktionalität der Benutzeroberfläche bereitstellen. Sie stehen ebenfalls weitestgehend für sich, haben also kaum Abhängigkeiten zur ersten Gruppe.
3. Pakete, deren Klassen Hilfsfunktionalität bereitstellen. Auch diese sind frei von Abhängigkeiten zu den oberen beiden.

Im Anschluß werden die wichtigsten Pakete aus 1. und 2. vorgestellt.

1.2.1 .pned.core.model

In diesm Paket befinden sich die Schnittstellenklassen des Datenmodells. Es handelt sich hierbei um das von der Logik-Schicht verwendete Datenmodell, welches nicht von der Benutzeroberfläche benötigt wird. Das Paket beinhaltet keine Implementierungen.

1.2.2 .pned.core

In diesm Paket befinden sich je eine mögliche Implementierungen der Schnittstellenklassen aus `.pned.core.model`. Außerdem liegen hier die Ausnahmen (`exceptions`), die in Klassen dieses Pakets auftreten können.

1.2.3 .pned.gui.core.model

Analog zu `.pned.core.model` befinden sich in diesem Paket diejenigen Schnittstellentypen, die für die Visualisierung notwendig sind, also etwa das zustandsabhängige Aussehen der Knoten und Kanten.

1.2.4 .pned.gui.core

Hier befinden sich Implementierungen der Schnittstellen aus `.pned.gui.core.model`.

1.2.5 .pned.gui.components

In diesem Paket und seinen Unterpaketen sind die eigentlichen Komponentenklassen abgelegt, also vor allem Erweiterungen von `javax.swing` Objekten für Knoten, Kanten und andere Elemente, außerdem das Hauptfenster, Dialogfenster, Menus, sowie Listener-Klassen welche den Arbeitsfluß der Benutzeroberfläche steuern. So gibt es zum Beispiel unter `.pned.gui.components.listeners` ein Listener-Objekt, welches für die Auswahl von Knoten zuständig ist.

1.2.6 .pned.control und .pned.gui.control

In diesen beiden Paketen und ihren Unterpaketen befinden sich die Klassen die für die eingangs geschilderte Art der Kommunikation durch Nachrichten zuständig sind. Diese Pakete bilden sozusagen das Herz des *PnEd*. Hier liegen jeweils drei Unterpakete für Ereignis- oder Nachrichtenarten:

.commands Hier liegen Nachrichten-Klassen, welche Veränderungen anstoßen, sozusagen *befehlen* sollen. In der Regel werden diese durch Nutzeraktionen angestoßen.

.events Hier liegen Nachrichten-Klassen, welche Ereignisse ausdrücken sollen, die nicht *befohlen* wurden, sondern vielmehr aufgrund vorangegangener *Befehle* enstanden sind, also aufgrund der Funktionsweise von Petri-Netz und Oberfläche ausgelöst wurden.

.requests Hier liegen Nachrichten-Klassen, welche Antworten erwarten. Derzeit gibt es nur zwei Nachrichten für die das zutrifft: 1. Die Anfrage nach einem Namen für ein neues Element, und 2. Die Anfrage nach einem Element für einen Namen. Letztere wird lediglich von der Oberflächenschicht benötigt.

In diesen Paketen liegen außerdem jeweils Listener-Klassen für die beschriebenen Ereignisse. Und schließlich liegen in den Paketen `.pned.control` und `.pned.gui.control` die Klassen `EventBus` und `PnEventBus` - dem eigentlichen Nachrichtenkanal.

Wie eingangs beschrieben, findet der Nachrichtenaustausch über den `EventBus` statt. Gültige Eingaben und Ereignisse die eine Veränderung beschreiben werden als ein Ereignis instanziert. dieses wird dann vom Aufrufer, z.B. einem Listener, an den `EventBus` weitergereicht. Dieser implementiert selbst die entsprechende Schnittstelle. Seine Aufgabe besteht nun lediglich darin, das Ereignis an die bei ihm registrierten Listener zu verteilen. Man könnte auch von einem Multiplexer sprechen.

Eine Konsistenzprüfung erfolgt in der Regel nicht. Es wird also nicht festgestellt, ob Ereignisse in einer bestimmten Reihenfolge bedient werden können. Auch eine Fehlerbehandlung wird nicht zurückgereicht. Das verwendete Protokoll ist sozusagen *verbindungslos*. Das birgt natürlich potentielle Probleme, war aber eine bewußte Entscheidung zugunsten looser Kopplung. Die Darstellung ist letztlich nicht für die Konsistenz des Datenmodells verantwortlich und umgekehrt!

1.2.7 Weiter Pakete

Es folgt lediglich eine Kurzbeschreibung der restlichen Pakete.

.concurrent Hilfsklassen für die Behandlung von Nebenläufigkeit.

.pned.gui.actions Unterklassen von `javax.swing.AbstractAction`, welche Nutzereingaben abstrahieren.

.pned.io Klassen, die für die Ein-, Ausgabe von Petri-Netzen zuständig sind. Hierzu gehört der vorgegebene `PNMLParser` sowie Klassen zur Serialisierung des Petri-Netz Modells.

.pned.util Hier liegen Hilfsklassen, die für den Betrieb des *PnEd* nicht nötig sind, aber während der Entwicklung hilfreich waren, z.B. ein Logger.

.swing Eine Ansammlung von eigenständigen Erweiterungen des `swing` frameworks. Hier liegt z.B. ein selbst-validierendes Eingabefeld, wie es im Editor verwendet wird.

.util Weitere, allgemeine Hilfsklassen.

2 Bedienung des *PnEd*

2.1 Verwaltung der Dateien

2.1.1 Erstellen eines neuen Petri-Netzes

Um ein neues Petri-Netz zu erstellen, wählen Sie in der Menüleiste ⇒ **File** ⇒ **New**. Diese Aktion verwirft das derzeit editierte Netz unwiderruflich und ohne Nachfrage.

2.1.2 Import/Öffnen

Zum Öffnen einer bereits vorhandenen Datei wählen Sie in der Menüleiste ⇒ **File** ⇒ **Import**. Wählen Sie die zu öffnende Datei aus Ihrem Dokumentenordner und bestätigen Sie über den **Open** Druckknopf. Die voreingestellte Dateiendung ist `.pnml`, es lassen sich jedoch beliebig benannte Dateien öffnen. Geladen werden können Dateien eines nicht näher spezifizierten PNML-Formats. Kann eine Datei nicht geladen werden, bleibt die Aktion folgenlos. Auch das Öffnen einer einer Datei verwirft das derzeit editierte Netz unwiderruflich und ohne Nachfrage.

2.1.3 Export/Speichern

Erstellte Petri-Netze können Sie exportieren über Menüleiste ⇒ **File** ⇒ **Export**, der Anwahl des Speicherortes und Bestätigen über den **Save** Druckknopf. Die Dateien werden in einem PNML-Format abgespeichert. Wird eine existierende Datei als Ziel des Speichervorgangs ausgewählt, so wird diese ohne weitere Nachfrage überschrieben. Es besteht keine Garantie, dass das gespeicherte Format eines zuvor geladenen Netzes binärkompatibel ist, nicht einmal wenn dieses Netz mit dem *PnEd* erstellt wurde.

2.2 Erstellen/Löschen von Knoten und Kanten

Innerhalb des Rasterfelds können Sie beliebig erweiterbar Knoten und Kanten setzen, das editierbare Feld vergrößert sich automatisch mit dem Setzen von Knoten bei Bedarf. Die Vergrößerung des Rasterfeldes geschieht ausschließlich auf diese Weise, also wenn neue Knoten in der Nähe des rechten oder unteren Randes gesetzt werden. Über die Bildlaufleisten kann dann der Sichtbare Bereich verändert werden.

2.2.1 Erstellen und Platzieren von Stellen

Zum Erstellen einer Stelle wählen Sie in der Menüleiste ⇒ **Edit** und aktivieren das Optionsfeld vor **Create place**, so dass das Suffix **(default)** erscheint. Alternativ können Sie ein Kontextmenü über einen Rechtsklick im Rasterfeld aufrufen und diese Einstellung dort vornehmen. Wenn Sie anschließend mit der linken Maustaste in das Rasterfeld klicken erhalten Sie automatisch eine Stelle.

Solange das Optionsfeld bei **Create place** gesetzt bleibt, generiert jeder Linksklick im Rasterfeld eine neue Stelle. Dies ist auch der voreingestellte Wert, so dass bei neu geöffnetem *PnEd* zunächst Stellen erzeugt werden.

2.2.2 Erstellen und Platzieren von Transitionen

Zum Erstellen einer Transition wählen Sie in der Menüleiste ⇒ **Edit** und aktivieren das Optionsfeld vor **Create transition**, so dass das Suffix **(default)** erscheint. Alternativ können Sie ein Kontextmenü über einen Rechtsklick im Rasterfeld aufrufen und diese Einstellung dort vornehmen. Wenn Sie anschließend mit der linken Maustaste in das Rasterfeld klicken erhalten Sie automatisch eine Transition.

Solange das Optionsfeld bei **Create transition** gesetzt bleibt, generiert jeder Linksklick im Rasterfeld eine neue Transition. Neu erzeugte Transitionen werden grün dargestellt, wodurch ihre Aktivierung kenntlich ist.

2.2.3 Verschieben von Stellen und Transitionen

Einzelne Stellen oder Transitionen können jederzeit mit dem Mauszeiger bei gehaltener linker Maustaste verschoben werden.

Zum Verschieben mehrerer Knoten ziehen Sie bei gehaltener linker Maustaste ein Auswahlfeld über die zu verschiebenden Stellen bzw. Transitionen. Die Knoten sind dann durch eine rote Umrahmung markiert. Ziehen sie nun einen beliebigen Knoten mit dem Mauszeiger bei gehaltener linker Maustaste, um alle ausgewählten Knoten zu verschieben.

2.2.4 Löschen von Stellen und Transitionen

Zum Löschen von Knoten ziehen Sie bei gehaltener linker Maustaste ein Auswahlfeld über die zu löschenden Stellen bzw. Transitionen. Die Knoten sind dann durch eine rote Umrahmung markiert. Wählen Sie über die Menüleiste oder das Kontextmenü ⇒ **Edit** ⇒ **Remove selected nodes** - die markierten Objekte werden ohne Nachfrage und unwiderruflich gelöscht.

Beim Löschen von Knoten werden automatisch auch die an den Knoten verankerten Kanten entfernt.

2.2.5 Erstellen von Kanten

Um eine Kante zu erstellen klicken Sie mit der linken Maustaste zweimal hintereinander auf einen Knoten. Es erscheint eine Kante, die Ihrem Mauszeiger folgt. Eine Rotfärbung kennzeichnet, dass die Kante am Ort Ihres Mauszeigers nicht verankert werden kann, eine Grünfärbung, dass eine Verankerung möglich ist.

Wenn Sie während des Erstellens einer ungültigen Kante mit der linken Maustaste klicken, wird die Kante verworfen. Wenn sie auf einem gültigen Zielknoten zweimal hintereinander klicken, wird die Kante am Zielknoten verankert.

2.2.6 Löschen von Kanten

An einem Knoten können alle eingehenden und alle ausgehenden Kanten gelöscht werden. Eine Auswahl und Löschung einzelner Kanten ist nicht möglich.

Um eingehende Kanten an einem Knoten zu entfernen, ziehen Sie bei gehaltener linker Maustaste ein Auswahlfeld über die Stellen bzw. Transitionen deren eingehende Kanten entfernt werden sollen. Die Knoten sind dann durch eine rote Umrahmung markiert. Über die Menüleiste oder das Kontextmenü wählen sie ⇒ **Edit** ⇒ **Remove incoming edges**. Die eingehenden Kanten werden entfernt.

Um ausgehende Kanten zu entfernen, verfahren sie wie oben und wählen stattdessen ⇒ **Edit** ⇒ **Remove outgoing edges**. Die ausgehenden Kanten werden entfernt.

2.3 Markieren von Stellen

Neu erstellte Stellen sind ohne Markierung, bzw. mit einer sogenannten Nullmarke versehen. Um einer Stelle eine Markierung zuzuweisen, klicken Sie mit der rechten Maustaste in eine Stelle. Ein kleines Textfeld erscheint in welches Sie eine Zahl eingeben können. Durch Bestätigen mit der `Enter` oder `Return` Taste wird der Stelle diese Markierung zugewiesen. Die Eingabe kann mit der `ESC` Taste abgebrochen werden. Eine ungültige Eingabe wird durch roten Text gekennzeichnet.

Die Markierung einer Stelle wird entweder durch einen schwarzen Punkt oder eine Zahl `>2` gekennzeichnet. Nullmarken werden nicht ausgezeichnet.

2.4 Aktivierung von Transitionen

Aktivierte, grün gefärbte Transitionen können geschaltet werden. Das Schalten erfolgt durch Klicken der rechten Maustaste auf der aktivierten Transition. Transitionen können nur einzeln geschaltet werden.

2.5 Beschriftung von Knoten

Um eine Stelle oder Transition zu beschriften, klicken Sie mit der rechten Maustaste in das Beschriftungfeld über einem Knoten. Es erscheint ein editierbares Textfeld in das Sie einen neuen Namen schreiben können. Durch Bestätigen mit der `Enter` oder `Return` Taste wird dem Knoten der neue Name zugewiesen. Die Eingabe kann mit der `ESC` Taste abgebrochen werden. Eine ungültige Eingabe wird durch roten Text gekennzeichnet.

Die Beschriftungen können mit gehaltener linker Maustaste relativ zu ihren Knoten verschoben werden.

2.6 Ändern der Größe von Knoten, Marken und Kanten

Um die Größe der Knoten, Schriftgröße der Markierungen oder die Größe der Pfeilspitzen zu verändern, wählen Sie in der Menüleiste ⇒ **Preferences** ⇒ **Settings**. Sie können nun über die Eingabe im Textfeld oder Verschieben des Reglers die jeweilige Größe ändern.

3 Bekannte Probleme

3.1 OutOfMemoryException bei NodeRequests

Requests - Ereignisse die eine Antwort erwarten und Asynchron gestartet werden, Überfluten den Threadpool. Diese Problem tritt bereits bei relativ kleinen Netzen auf, da die voreingestellte Lebensdauer abgehandelter Threads recht hoch ist. Pro Listener Aufruf wird ein neuer Thread gestartet, der weiterlebt nachdem der Aufruf (erfolglos) endet. Das Problem ist lösbar, indem ein eigener Threadpool implementiert wird, welcher Threads wiederverwendet, und beendete Threads aufräumt. Die Stelle im Code: `de.markusrother.pned.gui.control.PnEventBus.requestNode(NodeRequest)`