Dokumentation PetriNetEditor

Inhaltsverzeichnis

1. Grundidee, Konzepte, Designentscheidungen

1.1. Grundidee und Konzepte

Die Grundidee für die GUI war, dass der Anwender beim Bearbeiten eines Workflow-Netzes möglichst vieles kontextsensitiv erledigen kann. Er sollte also in der Lage sein, Sachen direkt anzuklicken und dann die möglichen Befehle vorfinden. Beispiele:

- Mit der rechten Maustaste auf eine leere Stelle klicken und dann über das Kontextmenü genau dort einen Knoten erzeugen.
- Eine aktive Transition mit der rechten Maustaste anklicken und schalten.

Außerdem habe ich versucht, übliche Tastaturkürzel einzusetzen. Beispiele:

- STRG+O zum Öffnen, STRG+S zum Speichern von Dateien oder F2 zum Umbenennen.

Hauptkonzept ist *Model-Controller-View.*

Es gibt mehrere Controller und Modelle (teilweise je 1 Instanz pro geöffneter Datei), die in der weiter unten folgenden Klassenaufteilung grafisch dargestellt sind:

- ApplicationController
- I18NManager
- DataModelController
  - DataModel und ValidationMessagePanel
- ValidationController
- GuiModelController
  - GuiModel
  - DrawPanel
    - MyMouseAdapter
- ActionManager

Mit den DrawPanels werden die Views für die einzelnen Dateien umgesetzt. (Es gibt somit, von der PNML-Datei abgesehen, nur 1 einzige Sicht auf die Daten.)

Natürlich habe ich auch versucht, andere Konzepte wie die Kapselung der Daten in den Modellen so weit möglich umzusetzen.

1.2. Weitere Designentscheidungen

Die Daten eines Netzes stehen in zwei Modellen, dem Daten- und dem GUI-Modell. Das Daten-Modell enthält die persistenten Daten, also alles, was in PNML-Dateien gespeichert wird. Das GUI-Modell zusätzliche Daten für die Darstellung auf der Zeichenfläche.

Für die eindeutige Zuordnung der Modelle zueinander (und zur PNML-Datei) verwende ich den eindeutigen kanonischen Pfadnamen: `file.getCanonicalPath()`

Dieser „modelName“ ist allen Objekten hinterlegt, die zu einer Datei gehören:

- DataModel und ValidationMessagePanel
- GuiModel und DrawPanel

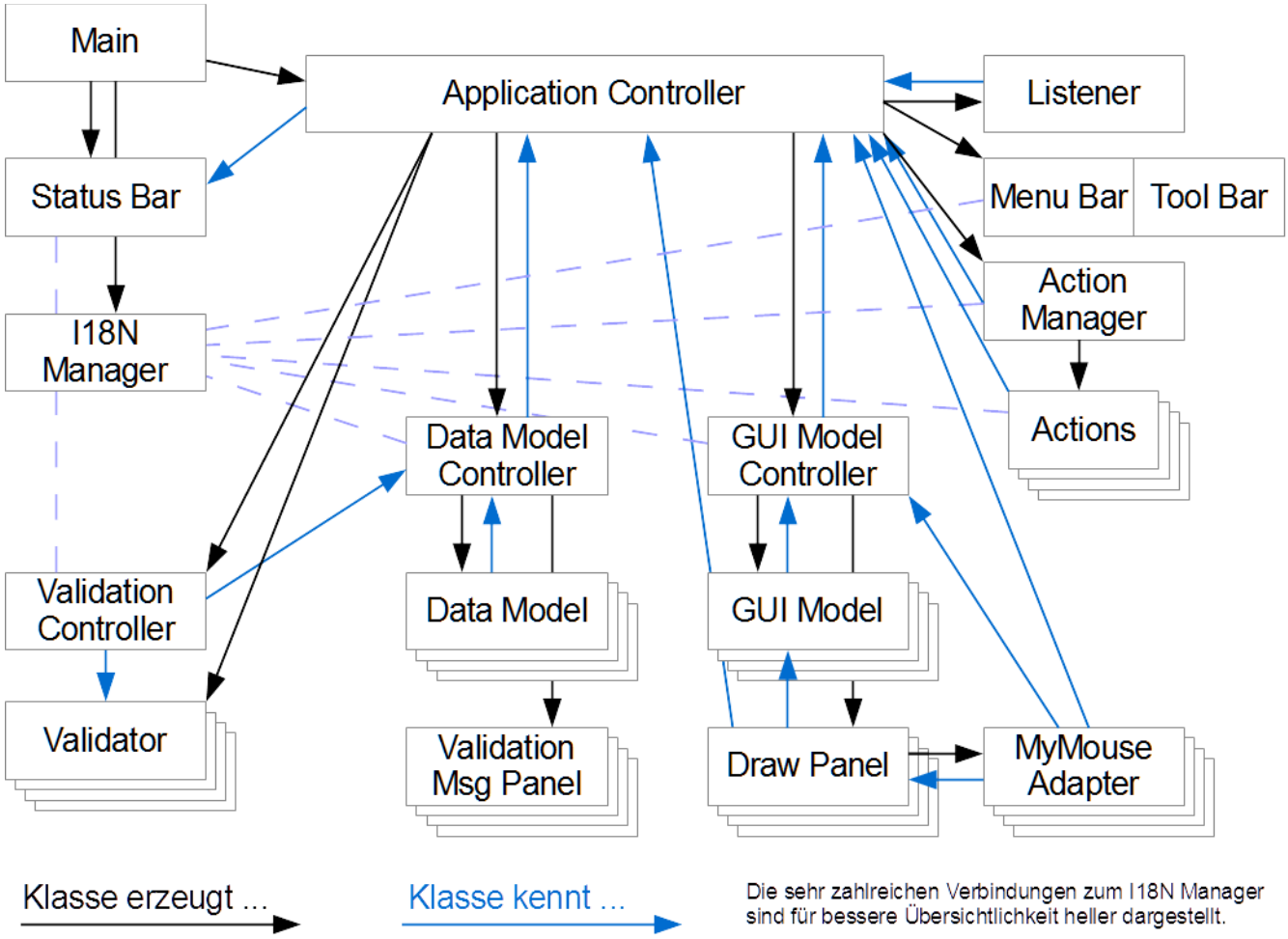
Ausnahmen sind:

- ValidationController, der stets den DataModelController nach dem aktuellen Model befragt.
- MyMouseAdapter: Kennt nur das DrawPanel und die darüberliegenden Controller.

Data- und GuiModelController verwalten je eine Map der geladenen Modelle mit „modelName“ als Schlüssel.

Der ApplicationController „verwaltet“ (mit Hilfe des TabListener) die derzeit aktive Datei, sorgt also dafür, dass andere Controller bei Bedarf zu einem anderen Modell wechseln.

1.3. Die Klassenaufteilung



Aufgaben der Packages

de.lambeck.pned.application

Klasse	Hauptaufgaben
`Main`	• Startparameter (Locale), Erzeugen des i18n-Managers (Lokalisierung) • Eigentlicher Programmstart (Aufruf des `ApplicationController`)
`AbstractApplication-Controller`	• Abstrakte Basisklasse für `ApplicationController` • Implementiert die Schnittstellen für die Verwendung der Statusleiste. • "Verwaltet" das `currentDirectory`, also das letzte vom User für das Öffnen oder Speichern verwendete Verzeichnis.
`ApplicationController`	• Erzeugen der weiteren Controller • Vorbereiten und Starten des `ValidatorController`-Threads • Aufbau der Programmoberfläche: • Im weiteren Programmverlauf: ◦ Koordination zwischen den Controllern, vor allem zwischen *Datenmodell- und GUI-Controller*

Klasse	Hauptaufgaben
	○ Verarbeiten von Bedienvorgängen (z.B. Menübefehle)
`TabListener`	Überwacht das Umschalten zwischen Dateien (Tab-Wechsel) durch den User.
`ComponentResizeListener`	Überwacht Größenänderungen des Programmfensters
`interface IInfo_Status ...`	Zentrale Schnittstellen für die Verwendung der Statusleiste durch andere Komponenten des Programms

de.lambeck.pned.application.actions

Klasse	Hauptaufgaben
`ActionManager`	Initialisieren der programmweit nutzbaren Action-Objekte (z.B. für `FileOpen`, `FileSave`) Aktualisierungen an den Action-Objekten ("enabled" state)
`interface IActionManager`	Für den öffentlichen Zugriff auf den `ActionManager`.
`AbstractPNAction`	Abstrakte Basisklasse für die konkreten Action-Objekte
`FileNewAction ...`	Konkrete Objekte vom Typ `AbstractPNAction` zur Verwendung als Schaltflächen in Menü- und Symbolleiste oder Kontextmenüs

de.lambeck.pned.elements

Klasse	Hauptaufgaben
`interface IElement ...`	**Schnittstellen** für gemeinsame Eigenschaften von Daten- und GUI-Modell, z.B.: • `interface IElement: String getId()` • `interface Itransition: boolean isEnabled()`
`enum EPlaceToken ...`	**Enums** für gemeinsame Eigenschaften von Daten- und GUI-Modell, z.B.: • `enum EnodeType: PLACE, TRANSITION` • `enum EplaceToken: ZERO(0), ONE(1)`

de.lambeck.pned.elements.data

Klasse	Hauptaufgaben
	Elemente im Daten-Modell: • DataElement: alle Elemente des Daten-Modells, mit Subtypen: ○ DataArc (Kanten) ○ DataNode (Knoten), wiederum mit Subtypen: ▪ DataPlace (Stellen) ▪ DataTransition (Transitionen) Die **Schnittstellen** definieren Methoden, die nur im Daten-Modell eine Rolle spielen, z.B.: • `interface IdataTransition: boolean checkEnabled()` Weil die Transition im Daten-Modell ihren "enabled"-Status selbst prüft, während es im GUI-Modell einen Setter gibt.
`abstract class DataElement,` `abstract class DataNode`	Stellen Eigenschaften und Methoden bereit, die für mehrere oder alle Typen von Elementen gelten, z.B: • `abstract class DataElement:` ○ `protected String id, public String getId()` • `abstract class DataNode:` ○ `protected String name` ○ `public String getName(), public void setName(...)` ○ `protected Point position` ○ `public Point getPosition(), public void setPosition(...)`
`class DataPlace, class DataTransition, class DataArc`	• Implementieren konkrete Elemente auf der Seite des Daten-Modells. • Speichern alle persistenten Daten, also Daten für PNML-Dateien.

de.lambeck.pned.elements.gui

Klasse	Hauptaufgaben
`class GuiPlace, class GuiTransition, class GuiArc`	**Elemente im GUI-Modell:** • Analog zu den Elementen im Daten-Modell: GuiElement, GuiArc, GuiNode, GuiPlace und GuiTransition Die **Schnittstellen** definieren analog Methoden, die nur im GUI-Modell eine Rolle spielen,

Klasse	Hauptaufgaben
	zum Beispiel: • `interface IGuiTransition`: ◦ `void setEnabled(boolean newState)` Weil hier der "enabled"-Status übermittelt wird nachdem das Gegenstück im Daten-Modell diesen Status bestimmt hat.

de.lambeck.pned.exceptions

Klasse	Hauptaufgaben
	Eigene Exceptions, bspw. für den Versuch, ein Duplikat eines Elements zu erzeugen. (Eine DuplicateException o.ä. habe ich erstaunlicherweise im Java-Standard nicht gefunden.)

de.lambeck.pned.filesystem

Klasse	Hauptaufgaben
`class FSInfo`	Hilfsklasse zum Bestimmen von Pfaden und Dateinamen, Prüfung, ob Dateien bereits existieren oder schreibgeschützt sind und ein "Datei Speichern unter..."-Dialog

de.lambeck.pned.filesystem.pnml

Klasse	Hauptaufgaben
`PNMLParser, PNMLWriter`	Parser und Writer für PNML-Dateien. Basiert auf den mit der Aufgabenstellung übergebenen Beispielen – insbesondere der Parser wurde aber umgeschrieben/erweitert.
`enum EPNMLParserExitCode`	Die möglichen ExitCodes, die der Parser nach dem Einlesen an den DataModelController zurückgibt. Dieser prüft dann in `private boolean acceptModel()` in Abhängigkeit von diesem Exitcode, ob die eingelesee Datei akzeptiert wird.
`enum EPNMLElement`	Wird im Parser verwendet, um temporär zu speichern, was für ein Element gerade aus den eingelesenen zeilen "zusammengebaut" wird.

de.lambeck.pned.gui

Klasse	Hauptaufgaben
`enum EcustomColor`	Enum für Farben mit "sprechenden" Namen

de.lambeck.pned.gui.icon

Klasse	Hauptaufgaben
`class ImageIconCreator`	Erzeugen von Icon-Objekten für die AbstractActions
`enum EiconSize`	Passende Größen-Parameter für `ImageIconCreator.getScaledImageIcon()`

de.lambeck.pned.gui.menuBar

Klasse	Hauptaufgaben
`class MenuBar, MenuCreator`	Implementiert die Menüleiste für das Hauptfenster + Hilfsklasse MenuCreator

de.lambeck.pned.gui.popupMenu

Klasse	Hauptaufgaben
	Mögliche Kontextmenüs für die Zeichenfläche (Drawpanel), Jeweils 1 Klasse für freie Flächen, Stellen, Transitionen und Kanten

de.lambeck.pned.gui.settings

Klasse	Hauptaufgaben
`class SizeSlider`	Implementiert den Schieberegler für die Größe der Elemente in der Symbolleiste

de.lambeck.pned.gui.statusBar / toolBar

Klasse	Hauptaufgaben
	Implementieren die Statusleiste und die Symbolleiste für das Hauptfenster.

de.lambeck.pned.i18n

Klasse	Hauptaufgaben
`class I18NManager`	Liefert lokalisierte Nachrichten und Bezeichnungen aus den ResourceBundles.
`class MnemonicString`	Hilfsklasse: Implementiert Objekte für Schaltflächen mit Name + "Accelerator key".

de.lambeck.pned.models

Klasse	Hauptaufgaben
`interface IModel`	Interface für gemeinsamen Eigenschaften und Methoden aller Elemente eines Petri-Netzes

de.lambeck.pned.models.data

Klasse	Hauptaufgaben
	Schnittstellen und Implementierung für das Daten-Modell und Daten-Modell-Controller

de.lambeck.pned.models.data.validation

Klasse	Hauptaufgaben
`IValidationController, ValidationController`	Der Validation Controller (Thread) und Schnittstelle für den Zugriff darauf
`abstract class AbstractValidator`	Abstrakte Basisklasse für Validatoren
`class ...Validator`	Die Validatoren: Start-, EndPlaces-, AllNodesOnPaths-, InitialMarking-, EnabledTransitions
`interface IValidationMsg, class ValidationMsg`	Implementieren die von den Validatoren gespeicherten Meldungen.
`enum EValidationResultSeverity`	Enum für die möglichen Level dieser Meldungen (wie schwerwiegend)

de.lambeck.pned.models.gui

Klasse	Hauptaufgaben
	Schnittstellen und Implementierung für das GUI-Modell und GUI-Modell-Controller

de.lambeck.pned.util

Klasse	Hauptaufgaben
`class NodeInfo`	Hilfsklassen für Infos zu Knoten (verwendet bei den Validatoren)
`class ConsoleLogger`	Hilfsklasse für Konsolenausgabe für Debugging o.ä.

1.4. Ablauf beim Programmstart

Main

1. Prüft Startparameter (Locale)
2. Dann wird das Hauptfenster initialisiert.
3. Außerdem werden Instanzen von *I18NManager* (mit der richtigen Sprache) und *StatusBar* erzeugt.
4. Vor dem Anzeigen des Hauptfensters wird der *ApplicationController* erzeugt.

ApplicationController:

5. Er erweitert *AbstractApplicationController* (allgemeine Methoden).
6. Initialisiert eine Reihe von Listenern z.B.:
   - *ComponentResizeListener* (Größenänderungen)
   - sich selbst als *WindowListener*, *WindowStateListener* (Minimieren, Maximieren...)
7. Erzeugt die wichtigsten Controller:
   - *ActionManager* (Schaltflächen, Menübefehle...)
     - (Von dort holt er sich gleich die Maps mit den verfügbaren Actions ab.)
   - *DataModelController*
   - *GuiModelController*
   - *ValidationController*

   ```
   void addControllers()
   ```
8. Erzeugt Validatoren und fügt sie dem ValidationController hinzu.

   ```
   void addValidators(),
   validationController.addValidator()
   ```
9. Dann erzeugt er die TabbedPane für die Registerkarten für geöffnete Dateien in der ContentPane (also im zentralen Bereich) des Hauptfensters.

   Hier wird ein weiterer Listener (*TabListener*) für den Wechsel zwischen Dateien hinzugefügt.
10. Anschließend werden *MenuBar*, *ToolBar* und *StatusBar hinzugefügt*.
11. Als letztes wird der *Validator-Thread* gestartet.

1.5. Funktion des Validator-Threads

1. Der ValidationController („Controller“) wird vom Application Controller als Thread gestartet nachdem die verfügbaren Validatoren übergeben wurden.
2. Der Controller speichert Validatoren in einer Map. (Damit sind sie individuell per Name ansprechbar.)
3. In einem **Intervall von 1 s** wird nun der Data Model Controller nach dem `currentModel` gefragt.
4. Bei diesem wird abgefragt, ob Validierung nötig ist:

   ```
   !dataModel.isModelChecked()
   ```
5. Falls ja, wird **als erstes der Status zurückgesetzt**.

   ```
   setModelChecked(true)
   ```

   Anderenfalls würde eine Veränderung, die der Benutzer vornimmt **während die Validierung gerade läuft**, eventuell übersehen werden, weil der Controller das Model einen Sekundenbruchteil später als valide und gecheckt markiert **obwohl gerade eine weitere Veränderung** vorgenommen wurde!
6. Der Controller holt sich beim Data Model Controller eine Referenz auf das Validation Message Panel für das aktuelle Modell. (Damit die Meldungen bei der richtigen Datei angezeigt werden.)
7. Dann ruft der Controller nacheinander jeden seiner Validatoren auf und übergibt jeweils das aktuelle Modell als Parameter.

   ```
   startValidation(dataModel, ...)
   ```
8. Jeder Validator führt eine eigene Liste mit Nachrichten, die beim Validieren des Modells anfallen.
9. Nach Beenden des Validators fragt der Controller diese Nachrichten ab:

   ```
   validator.hasMoreMessages()
   validator.nextMessage()
   ```
10. Jede Nachricht hat einen „severity“-Status, also eine Aussage darüber, wie schwerwiegend die Meldung und ob das Modell insgesamt noch valide ist.

    Der insgesamt höchste (schwerwiegendste) Level aus

    ```
    enum EValidationResultSeverity
    ```

    ist dabei ausschlaggebend. Kommt also bspw. ein Validator zum Ergebnis ***WARNING***, kann das Gesamtergebnis durch andere Validatoren nur noch zu ***CRITICAL*** verschlechtert werden.

11. Jedem Level ist eine Farbe für das Message Panel zugeordnet, diese wird nun vom Validation Controller gesetzt:
    ```
    msgPanel.setBgColor()
    ```
12. Abschließend reicht der Controller das Ergebnis ans Modell weiter.
    ```
    dataModel.setModelValidity(isModelValid)
    ```

Manueller Validator-Aufruf

Der ValidationController kann manuell aufgefordert werden, eine spezielle Validierung „außer der Reihe“ für ein bestimmtes Modell durchzuführen.

Der Aufruf lautet:

```
requestIndividualValidation(validatorName, dataModel)
```

Verwendet wird er vom Data Model Controller nachdem eine Transition geschaltet wurde.

- Weil:
  - Nur genau 1 Validator benötigt wird.
  - Vor allem aber, weil andere Validatoren (insbesondere der StartPlaceValidator das Netz nicht in den Ausgangszustand zurücksetzen sollen.

1.6. Bekannte Probleme

Herausgabe von Referenzen auf Modelle und Elemente

(z.B. Liste der Elemente eines Modells) statt Kopien. Die Liste selbst wird zwar als Kopie erzeugt, die Elemente der Listen sind aber immer noch die originalen Referenzen. Was manchmal erwünscht ist, weil die erfragenden Methoden Änderungen an einzelnen Elementen vornehmen sollen. In anderen Fällen sollen diese Methoden eigentlich nur Lesen aber nichts ändern (dürfen). Dies würde sehr großen Aufwand für „DeepCopy“ erfordern, der mir deutlich zu groß erschien.

Weiteres:

- Manchmal funktioniert das Löschen eines Elements nur mit dem Menübefehl, aber nicht mit ENTF auf der Tastatur. Situation nicht nachstellbar, Ursache noch nicht gefunden.

1.7. Nicht umgesetzte Ideen

Zeichenfläche:

- Auswahlrahmen zum Markieren mit der Maus aufziehen und „Alles markieren“ mit STRG+A
- Bei Neustart der Simulation zum Anfangsknoten scrollen -> ScrollRect()?

Für Schreibgeschützte Dateien:

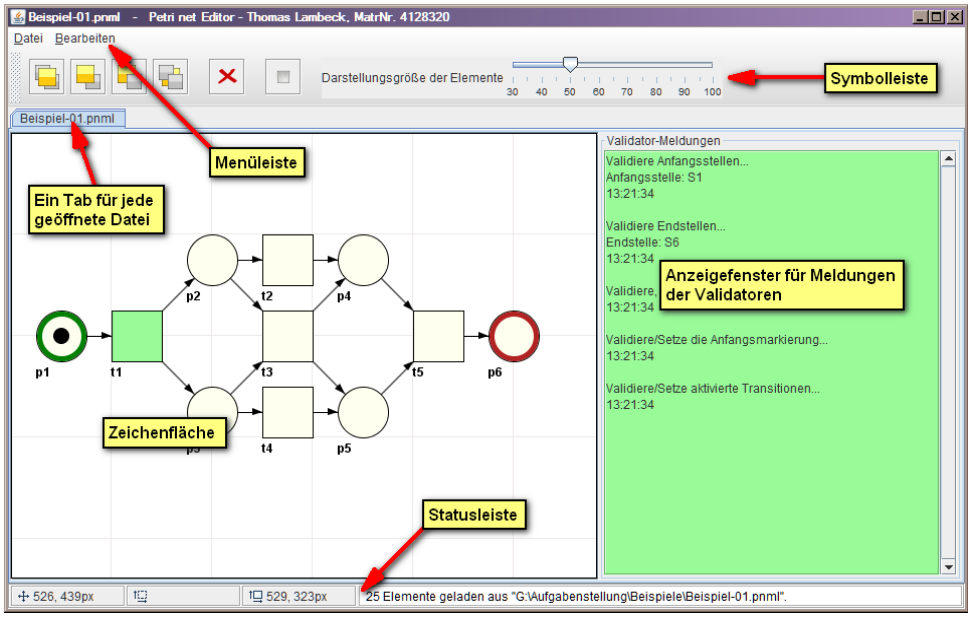
- Fehlermeldung ersetzen durch automatisches "Speichern unter..." beim Versuch, zu speichern
- Anzeige von "schreibgeschützt" in der Titelzeile

Rückgängig-Funktion:

- Das erschien mir wegen der bereits erwähnten Problematik mit „DeepCopy“ zu schwierig für die zur Verfügung stehende Zeit.

2. Bedienungsanleitung

2.1. Übersicht



Zeichenfläche und Anzeigefenster für Meldungen der Validatoren

Für jede geöffnete Datei gibt es eine Zeichenfläche und ein Anzeigefenster für Validator-Meldungen.

> Sind mehrere Dateien geöffnet, gibt es einen Tab für jede Datei (ähnlich typischen Internet-Browsern).

Menüleiste

Im **Menü "Datei"** finden Sie die Befehle:

- **"Neu"**: Erzeugt eine neue, leere Datei.
- **"Öffnen..."**: Anzeige des Dialogfelds "Öffnen" zur Auswahl einer Datei
- **"Schließen"**: Schließt die aktuelle Datei. Falls nötig, wird gefragt, ob gespeichert werden soll.
- **"Speichern"**, **"Speichern unter..."**: Speichern von Dateien (direkt bzw. nach Auswahl eines Dateinamens). Speichern schreibgeschützter Dateien wird selbstverständlich unterbunden.
- **"Beenden"**: Beendet das Programm direkt, falls keine Dateien mehr geöffnet oder alle gespeichert sind. Anderenfalls werden nur gespeicherte Dateien geschlossen und es folgt eine Abfrage zum Speichern.

Im **Menü "Bearbeiten"** finden Sie einige Befehle zum Bearbeiten und Simulieren der Workflow-Netze:

- "**Umbenennen...**": Anzeige eines Dialogfelds zur Eingabe eines neuen Namens. Nur möglich bei genau *1 einzeln markierten Element*. *Nur Knoten*, also Stellen und Transitionen, *können umbenannt werden*.
- "**Löschen**": Löscht alle markierten Elemente.

  > Beim Löschen von Knoten werden außerdem alle damit verbundenen Kanten gelöscht.
- "**Simulation beenden**": Setzt die Markierung eines Netzes zurück auf Anfangsmarkierung (Marke an Anfangsstelle). Die Simulation wird später beschrieben.

Menüs können **auch über die Tastatur** bedient werden. Die meisten Schaltflächen zeigen durch einen unterstrichenen Buchstaben an, welche Taste zu drücken ist.

> Menü "Datei": **ALT+D**, Befehl "Neu": **N**

Die meisten Befehle haben Hotkeys (hinter dem Namen angezeigt), die auch ohne Menü funktionieren.

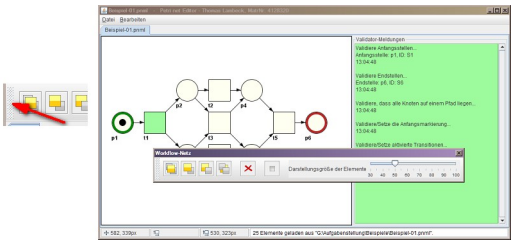
> Beispiel: "Öffnen": **STRG+O**

Symbolleiste

Hier finden Sie die Schaltflächen "Löschen" und "Simulation beenden" von der Menüleiste wieder. Außerdem:

- Schaltflächen, mit denen Sie ein markiertes Element über oder unter anderen anordnen können, falls sich Elemente überlappen.
- Einen Schieberegler zur Beeinflussung der Darstellungsgröße der Elemente.

Die Symbolleiste ist mit dem kleinen "Anfasser" auf der linken Seite frei verschiebbar.



Statusleiste

Zeigt Informationen zur Mausposition und zum "Platzverbrauch" Ihres Workflow-Netzes an.

Im rechten Teil befindet sich die Statuszeile, in der das Programm wichtige Statusmeldungen ausgeben kann.

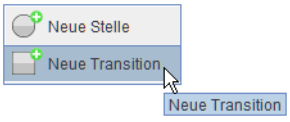
2.2. Bearbeiten von Workflow-Netzen

Der Validator arbeitet ständig im Hintergrund. Er wird also bereits beim ersten auf der Zeichenfläche angelegten Element versuchen, das Workflow-Netz zu validieren. (Weitere Informationen zu seiner Funktion folgen später.)

Ihr erstes Workflow-Netz

1. Legen Sie mit **"Neu" im Menü "Datei"** eine neue Datei an.
2. Legen Sie nun den ersten Knoten an.

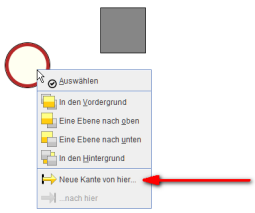
   Knoten werden über das **Kontextmenü der Zeichenfläche** erzeugt. Klicken Sie dazu an gewünschter Stelle mit der rechten Maustaste auf eine freie Stelle. Und wählen Sie *"Neue Stelle"* (Kreis) oder *"Neue Transition"* (Quadrat).



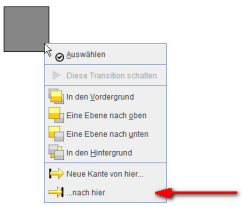
3. Legen Sie **zwei** Stellen und **eine** Transition an.

   Nun brauchen wir noch die **Kanten** (Pfeile). Diese können nur **zwischen unterschiedlichen Knoten** erzeugt werden. Sie benötigen also stets mindestens 1 Stelle und 1 Transition!

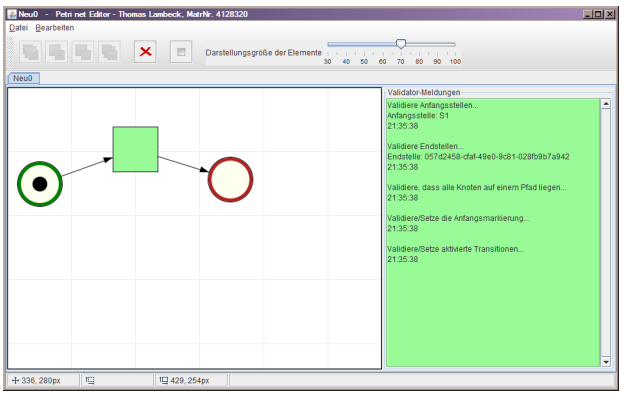
4. Klicken Sie mit der rechten Maustaste auf den Knoten, der Startpunkt der Kante sein soll und wählen Sie *"Neue Kante von hier..."*.



5. Klicken Sie nun ebenfalls mit der rechten Maustaste auf den Knoten, der Zielpunkt der Kante sein soll und wählen Sie *"...nach hier"*.



6. Legen Sie nun den zweiten Pfeil so an, dass ein Pfeil von einer der Stellen zur Transition hin und der andere davon weg zur anderen Stelle zeigt.



Ihr erstes Workflow-Netz ist fertig. (Der Validator hat es auch gleich validiert und die Marke an die Anfangsstelle gesetzt.)

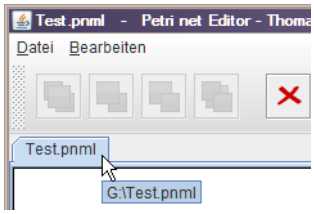
7. Speichern Sie die Datei mit *"Speichern unter..." im Menü "Datei"*.

   Dieses Dialogfeld sollte Ihnen durch andere Programme bekannt sein.

   Beim Aufruf zeigt es nur Dateien mit Endung ".pnml" an. Falls ihre Datei einen anderen Namen haben soll, wählen Sie im Dropdown-Menü für den Dateityp "Alle Dateien" aus.

8. Endet ihr Dateiname nicht auf ".pnml", schlägt das Programm außerdem vor, die Endung für Sie anzuhängen (was sie aber auch ablehnen können).

   Der Tab zeigt den neuen Namen der Datei an. Wenn Sie den Mauszeiger über den Tab halten, wird im Tooltip der komplette Pfad zur Datei angezeigt.



9. Analog: der *"Öffnen..."-Dialog*.

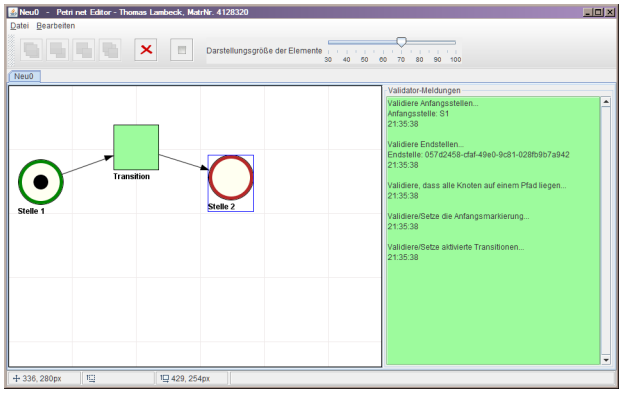
   Auch hier können Sie zwischen "Petri-Netz-Dateien (*.pnml)" und "Alle Dateien" wählen.

Knoten benennen, bewegen, löschen

10. Markieren Sie nun **einen** der Knoten durch Anklicken mit der linken Maustaste.
11. Wählen Sie nun *"Umbenennen..." im Menü "Bearbeiten"* (oder einfach *F2* auf der Tastatur) und geben Sie im Eingabefeld einen Namen (z.B. "Stelle 1") ein.

    Nur Knoten können benannt werden.

12. Benennen Sie auch die anderen Knoten.



13. Zum **Erweitern der Markierung** halten Sie beim Anklicken die *STRG-Taste* gedrückt. (Erneutes Anklicken mit STRG hebt eine Markierung wieder auf.)
14. **Markierte** Knoten können Sie mit der Maus **bewegen**. Halten sie dazu die linke **Maustaste gedrückt bis der Cursor zum Verschiebe-Cursor wechselt.**

    Die Zeitspanne entspricht der auf Ihrem Computer eingestellten Doppelklickgeschwindigkeit.

15. Zum Löschen eines Elements, markieren Sie es wieder mit einem Mausklick und wählen *"Löschen" im Menü "Bearbeiten"* (oder einfach *ENTF* auf der Tastatur).

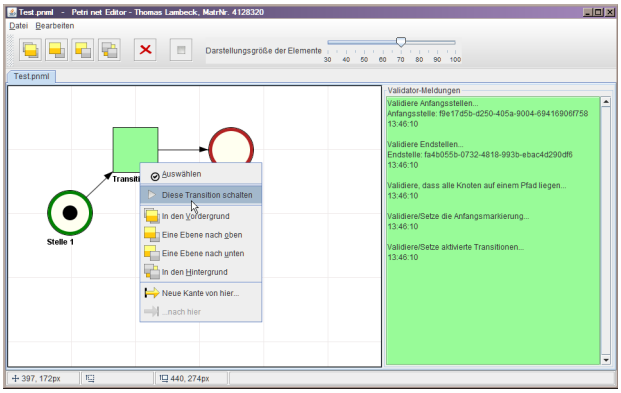
    Haben Sie mit STRG mehrere Elemente markiert, werden alle gemeinsam gelöscht.

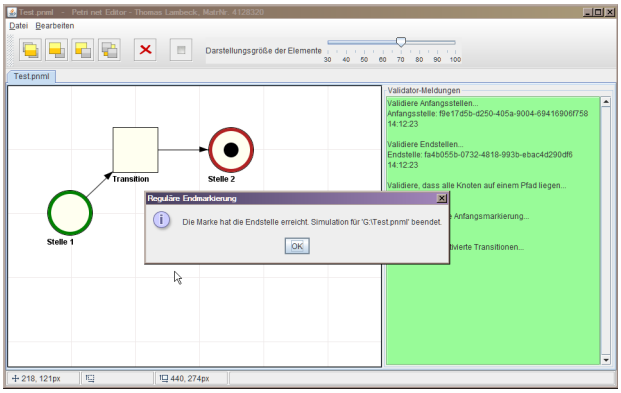
Validierung

Hinweis: Der Validator prüft nur **1x pro Sekunde** die jeweils aktuelle Datei, um Rechenleistung zu sparen. Es kann also nach einer Veränderung einen Augenblick dauern, bis er "reagiert".

Eine **laufende** (oder noch nicht erfolgreiche) **Validierung** wird durch **gelbliche Hintergrundfarbe** angezeigt.

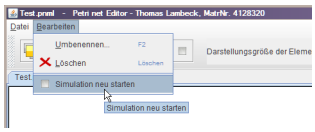
16. Der Validator hat erkannt: unsere *Transition ist aktiviert*. Signalisiert wird es durch die *grüne Füllfarbe*.
17. Wir können diese Transition nun schalten. Klicken Sie dazu mit der rechten Maustaste auf die Transition und wählen Sie *"Diese Transition schalten"*.





18. In unserem Fall ist die Simulation bereits beendet, weil die reguläre Endmarkierung erreicht ist.

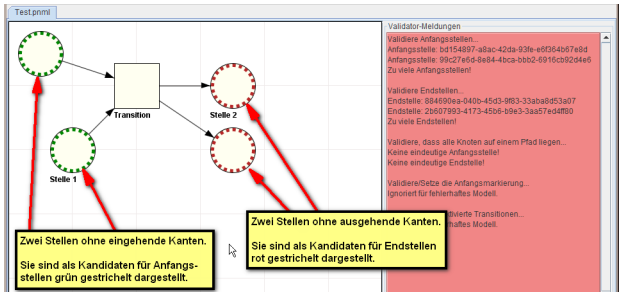
19. Sie können eine Simulation jederzeit (auch "mitten-drin") neu starten.



Vorhandene Validatoren

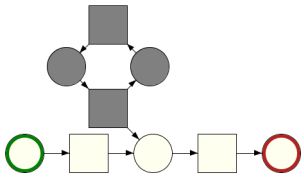
Prüfung 1: eindeutige Anfangs- und Endstelle? Hervorgehoben durch einen durchgezogenen **grünen** bzw. **roten** Kreis. (Siehe vorherige Abbildungen)

Gibt es aber mehr als 1 Stelle ohne ein- bzw. ausgehende Kanten, wird sie als möglicher *Kandidat* markiert, um zu verdeutlichen, wo noch etwas getan werden muss. Dies geschieht durch einen *gestrichelten grünen/roten Kreis*:



Das Benachrichtigungsfenster zeigt hier durch **rote Hintergrundfarbe** an, dass die **Validierung nicht erfolgreich** war.

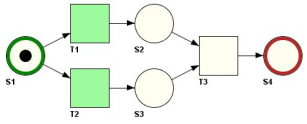
Prüfung 2: Alle Knoten liegen auf Pfaden zwischen Anfangs- und Endstelle? Alle anderen werden grau gefärbt:



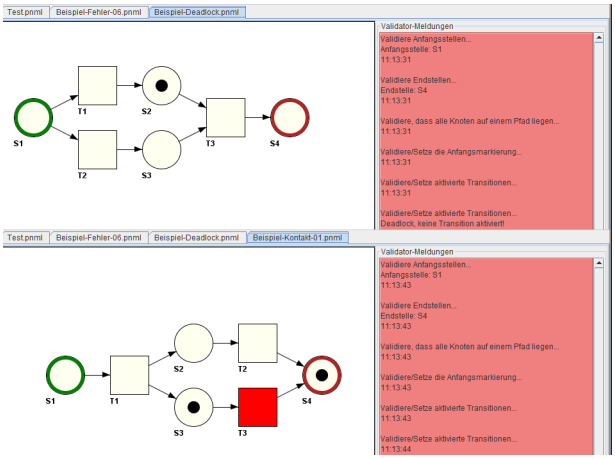
Prüfung 3: Hier wird die Anfangsmarkierung gesetzt. Das heißt, alle Marken werden entfernt und eine einzige auf die eindeutige Anfangsstelle gesetzt.

Ausnahme: Die Datei wurde gerade geöffnet: *Dann bleiben die Marken, wo Sie gespeichert wurden und sie können eine Simulation fortsetzen.*

Prüfung 4: Welche Transitionen können schalten? Diese (aktivierten) Transitionen werden hellgrün eingefärbt:



Außerdem prüft dieser Validator auf "Deadlocks" (keine Transition aktiviert) und unsichere Transitionen (Kontakt). Unsichere Transitionen werden rot eingefärbt:



Tipp: Können Sie eine Validator-Meldung nicht zuordnen, weil Sie Ihre Knoten noch nicht benannt haben, halten Sie den Mauszeiger über den Knoten. Im eingeblendeten Quicktipp werden Name und ID angezeigt: