GUI-Generierung mit EMUGEN

Dr. Alfons Brandl

Inhalt

GUI-Generierung mit EMUGEN im Überblick

Eingabe:

- ***Datentypen*** *(bestimmen GUI-Inhalt)*
- ***Dialogtyp*** *(GUI-Zustände und Aktionen)*
- ***Layout*** *(wird generisch festgelegt, kann aber im Speziellen überschrieben werden)*

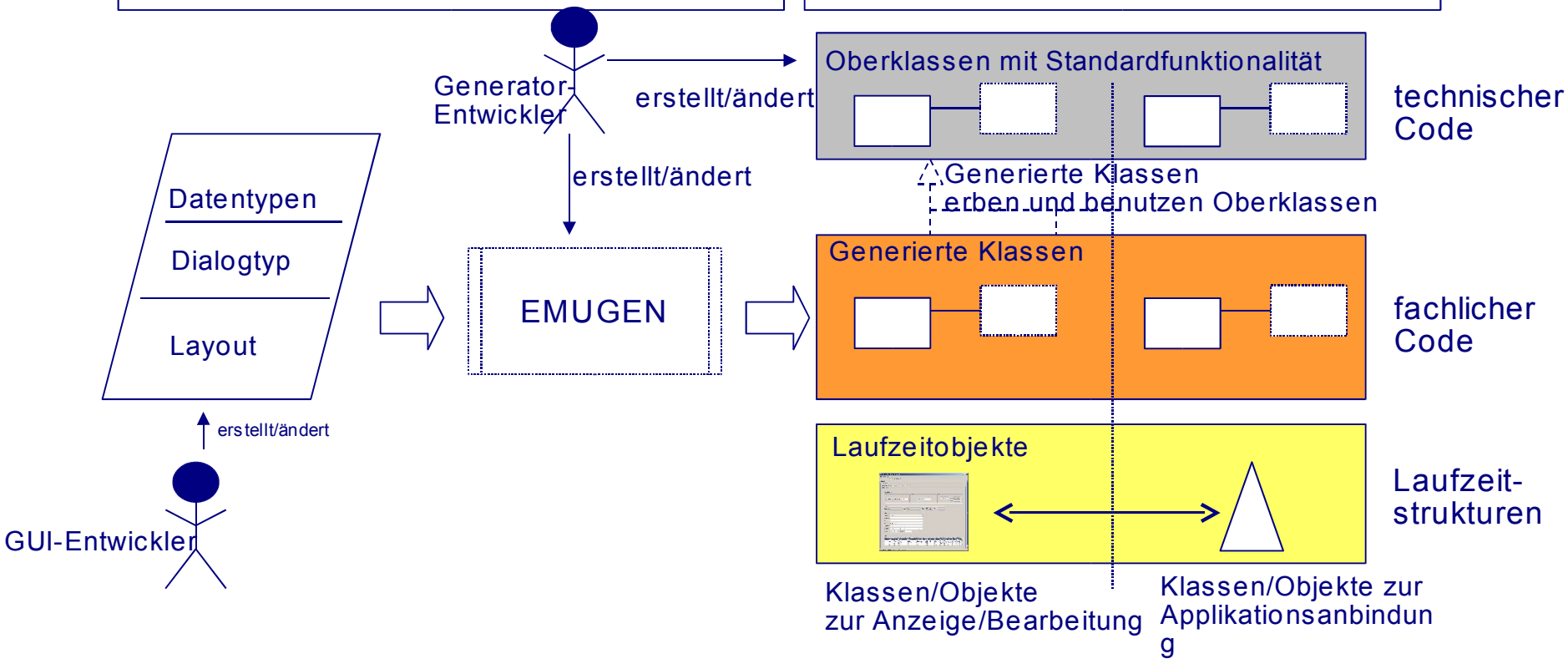
Eingabe-Notation sollte möglichst abstrakt sein (z.B. BNF, XML oder DB-Metainfos, manchmal kann auch eine geeignete API günstig sein)

GUI-Generator:

- *erzeugt die variablen Teile eines vorgefertigten objektorientierten Frameworks*
- *funktioniert wie ein einfacher Compiler*

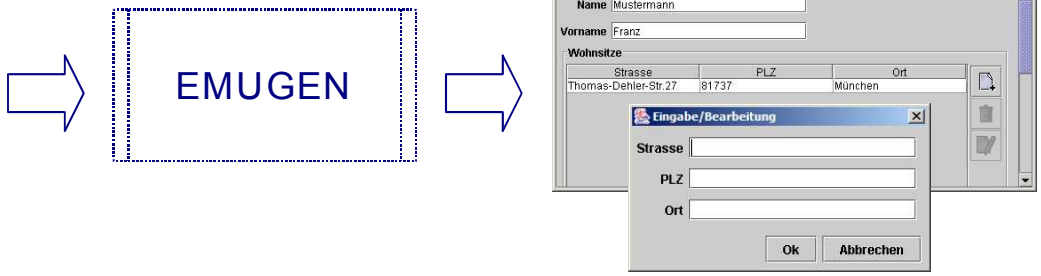
Vorgehensweise zur Entwicklung:

gehe immer von konkreten fachlichen Beispielen aus und versuche, diese mit dem GUI-Generator umzusetzen



Beispiel 1: Einfache GUI für Datentyp Person

```
Person::=String:Name
         String:Vorname
         Adresse*:Wohnsitze
Adresse::=String:Strasse
         String:PLZ
         String:Ort
```

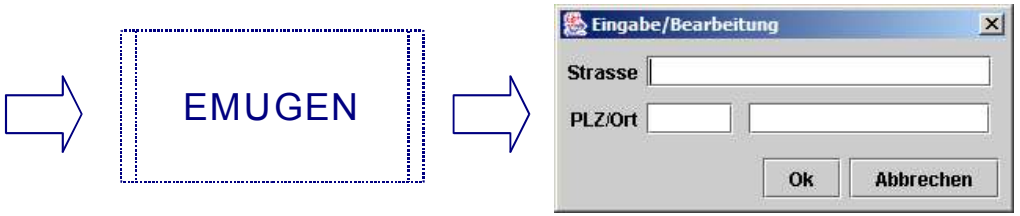


Mehr muss nicht angegeben werden!
(gesamte technische Funktionsweise kann durch **GUI-Generator und Oberklassen** festgelegt werden)

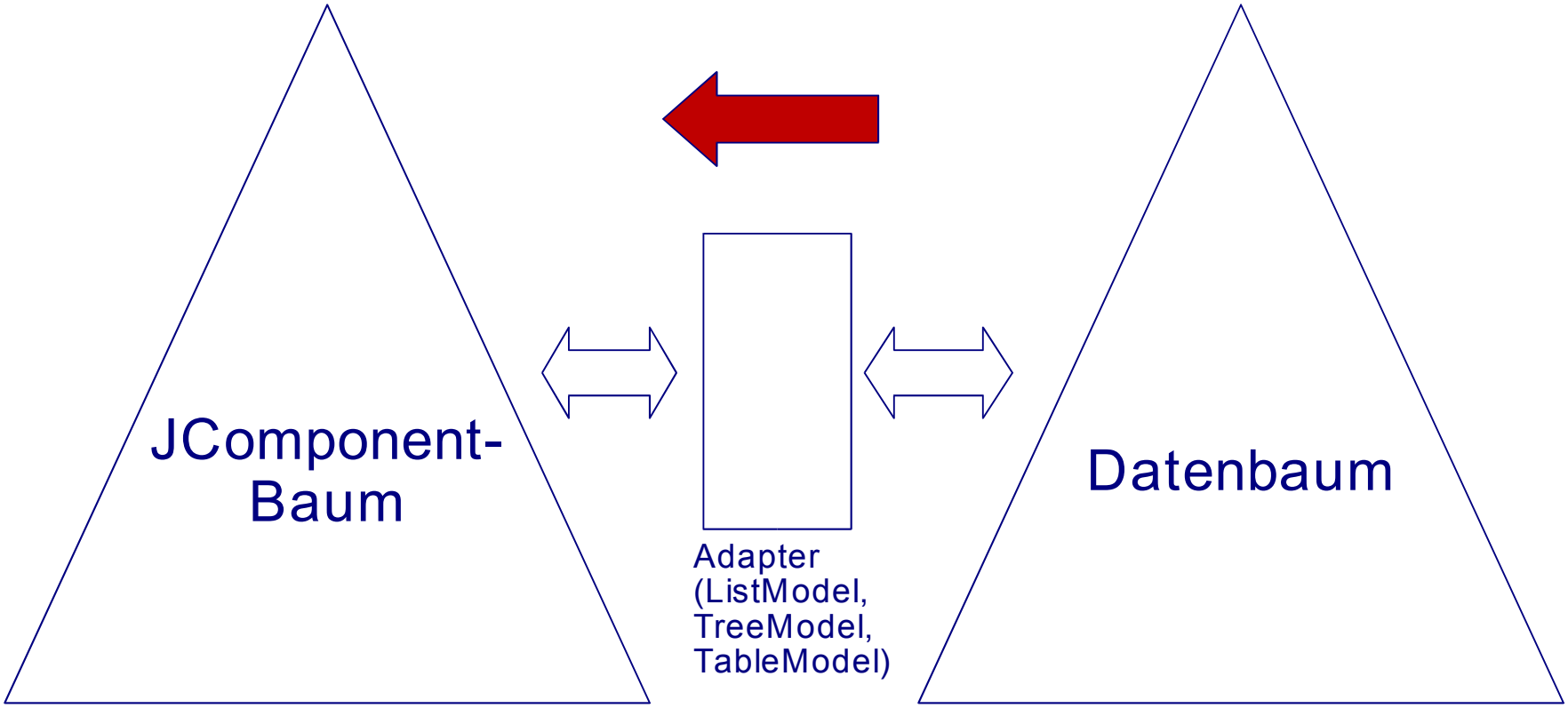
Wenn spezielles Layout, etwa für Adresse, gewünscht wird: Layout-Anpassung durch Verwendung einer AP

```
form layout of Adresse {:
 //entferne den Ort-Eintrag
 labels.removeElementAt(labels.size()-1);
 components.removeElementAt(components.size()-1);
 //integriere die Beschriftungen
 PLZLabel.setText("PLZ/Ort");
 PLZPanel.getTextField().setColumns(5);
 OrtPanel.getTextField().setColumns(14);
//Oberklassen sind im package emu_runtime
 JPanel PLZmitOrt=new emu_runtime.InLinePanel();
 PLZmitOrt.setLayout(new BorderLayout());
 PLZmitOrt.add(PLZPanel,BorderLayout.WEST);
 PLZmitOrt.add(OrtPanel,BorderLayout.CENTER);
 //ersetze PLZPanel mit dem zusammengesetzten Panel
 components.removeElementAt(components.size()-1);
 components.append(PLZmitOrt);
 super.formlayout();
:}
```

Anm.: Man könnte sich hier auch eine eigene Sprache/XML übe
durch Verwendung einer Java-API ist man flexibler,
aber implementierungsnäher, d.h. die Eingabe wird technischer

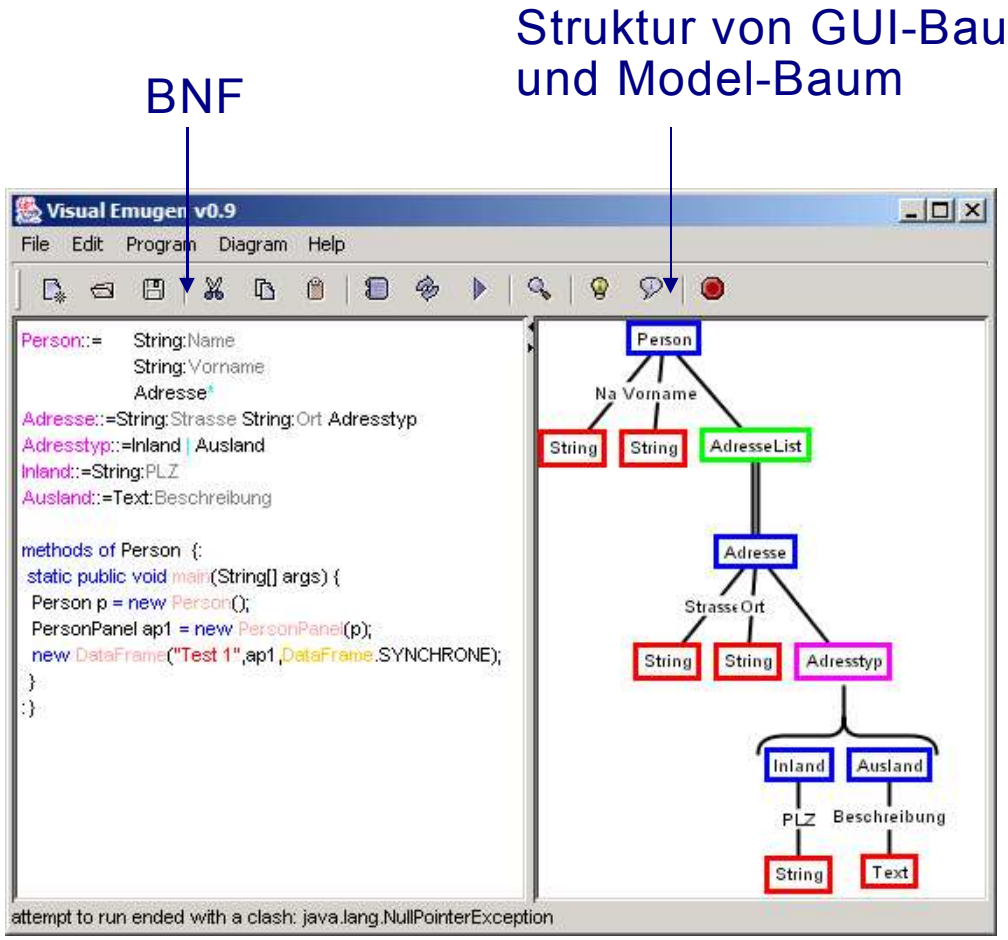


Grundlegende Bausteine: Detail zur Schnittstelle



Wie erstelle ich die Klassen für die beiden Java-Bäume?

1. *Beschreibe Baum durch BNF*
2. *Baue Generator, der daraus Klassen generiert mit Standardzugriff- und import/exportXML-Methoden)*
3. *Applikationsspezifischer Code kann in der Eingabe den Klassen hinzugefügt werden*
4. *Dadurch kann auch Default-Layout überschrieben werden (Java-Code in den GUI-Klassen)*



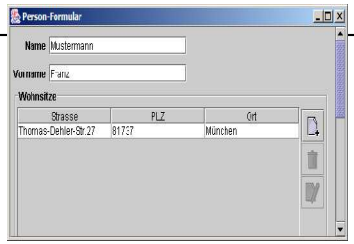
Zur Laufzeit

Herkömmlicher Weg

2. ***Erzeuge GUI-Baum (in Java: Component-Hierarchie)***
3. ***Fülle GUI-Baum mit Inhalt (bzw. stelle Verbindung zum Model her)***

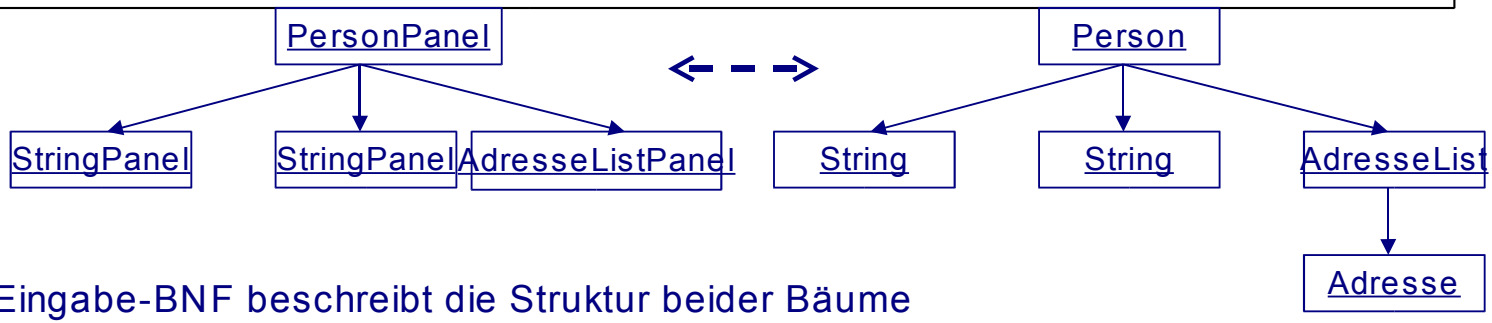
Bei EMUGEN

6. ***Erzeuge Daten-Baum und fülle ihn mit Inhalt***
7. ***Erzeuge GUI-Baum zum Daten-Baum***
8. ***Änderungen am Daten-Baum führen zu Änderungen am GUI-Baum***




```
new PersonPanel(p);
```

```
Person p = new Person();
```



die Eingabe-BNF beschreibt die Struktur beider Bäume

Was ist technisch zu beachten?

- ***Verwende objektorientierte Möglichkeiten***
  - **Beispiel-Generator verwendet das Visitor-Pattern**
  - **Scanner, Parser und die Klassen des abstrakten Syntaxbaums des Generators werden generiert (jflex, cup, classgen) und sind dadurch einfach erweiterbar (das Visitor-Interface wird ebenfalls generiert)**
  - **Das Layout kann auf konkreter Typebene (z.B. Adresse) oder auf Metaebene (z.B. für alle Varianten) definiert werden**
  - **Die API kann in den Oberklassen der Zielarchitektur erweitert werden auch ohne Veränderung des Generators (wurde hier gemacht, um die Muss- und Berechnungsfelder zu färben, solche Möglichkeiten sind die Vorteile einer API für das Layout)**
- ***Verwende Typsicherheit der Zielsprache um Fehler bei der komplexen Entwicklung zu vermeiden (d.h. auch: programmiere nicht jede Typsicherheit, die in der Zielsprache vorhanden ist, auch in den Generator)***

Bewertung

- *Vorteile*
  - Generelle Kritik am generativen Ansatz (zu unflexibel, zu kompliziert) sollte durch die gezeigte Funktionsweise mit den Beispielen abgeschwächt sein
  - die Möglichkeit, auch auf Metaebene zu arbeiten (siehe Beispiel 2, hier wurde das Layout aller Varianten im aktuellen Package festgelegt) führt zu einer effizienteren Entwicklung als mit Layout-Werkzeugen (wie z.B. VisualBasic)
  - Ansatz ermöglicht die effiziente, arbeitsteilige GUI-Entwicklung (die generierten Java-Klassen können in Packages organisiert werden und beliebig verwendet)
  - die kompakte, formale Beschreibung der Fachlichkeit (Datentypen, Aktionen) kann auch zur Dokumentation verwendet werden oder später, wenn das System auf einer anderen technischen Basis abläuft
  - GUI-Entwickler braucht weniger Wissen als bei direkter Anwendung von Swing (für 80% der Anwendungen)
- *Probleme*
  - GUI-Entwickler brauchen gewisses Abstraktionsvermögen
  - Generator-Entwickler braucht breites Wissen (Compilerbau+GUI)

EMUGEN-Eingabe

Datentyp-Notation:

- ***BNF-ähnlich***
- ***Metatypen: Tupel, Varianten, Listen, Referenzen, Grundtyp: String***

Dialogtyp-Notation: Beschreibungselemente für

- ***Zustandsautomaten und***
- ***Aktionen mit Anbindung an Java (vgl. Scannergeneratoren wie lex)***

Layout-Notation:

- ***bei EMUGEN direkt in Java (hat Vor- und Nachteile)***
- ***pro Typ und pro Metatyp kann spezielles Layout definiert werden***

Allgemein: Wie kommt man zu einem GUI-Generator?

1. *Schreibe fachliche Anforderungen an eine repräsentative Beispiel-GUI auf*
2. *Entwickle lauffähigen Prototyp für die Beispiel-GUI, der die fachlichen Anforderungen voll (!) erfüllt*
3. *Entwickle eigene Eingabenotation (je nach Projektanforderungen evtl. XML- oder DB-orientiert)*
4. *Beschreibe fachliche Anforderungen mit eigener Eingabenotation*
5. *Erstelle Framework für GUI-Generator*
6. *Erstelle Framework für Laufzeitarchitektur (welche Elemente werden generiert, welche wie z.B. bei EMUGEN durch Oberklassen bereitgestellt)*
7. *iteriere 3. bis 6. solange, bis das Beispiel generiert werden kann*
8. *wenn Beispiel generiert werden kann, dann teste weitere Beispiele unter Einbeziehung von Fachexperten*