Generieren vs. Interpretieren

Die andere Seite von MDSD

ex|Xcellent solutions

völter ingenieurbüro für softwaretechnologie

Inhalt

- Begriffsklärung
  - Generiert vs. Generisch
  - Gemeinsamkeiten
  - Generieren
  - Interpretieren
- Beispiel: Statemachine
  - Gemeinsamkeiten
  - Generierender Ansatz
  - Generischer Ansatz
- Praxisbeispiele
  - Typische Anwendungsszenarien
  - Konkrete Projektbeispiele
- Vergleich und Bewertung
- Fazit

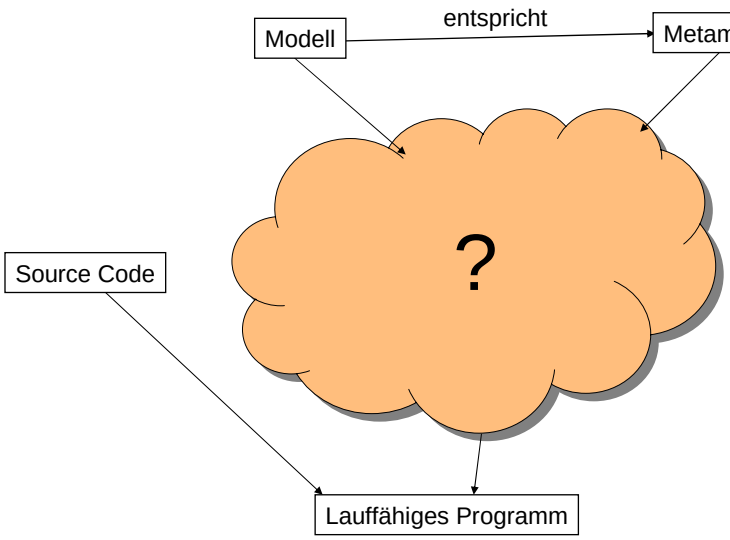
Generiert vs. Generisch

- *Generiert*
  - **Erzeugen** von Quelltext, Konfigurationsdateien und anderer textueller Artefakte aus einem Modell heraus
  - **Statisch**
  - Vor dem **Kompilieren**
  - Ergebnis: Viel (generierter) spezifischer Quelltext
- *Generisch*
  - **Interpretieren** von (Modell-)Informationen im Programm
  - **Dynamisch**
  - Zur **Laufzeit**
  - Ergebnis: Wenig (handgeschriebener) allgemeiner Quelltext

Gemeinsamkeiten

- **Abstraktion** des Problembereichs
- Definition eines **Metamodells**
- **Instantiierung** des Metamodells
- **Auswertung** des Metamodells
- Gleiches **Ziel**
  - Vermeidung wiederholter und manueller Programmiertätigkeit

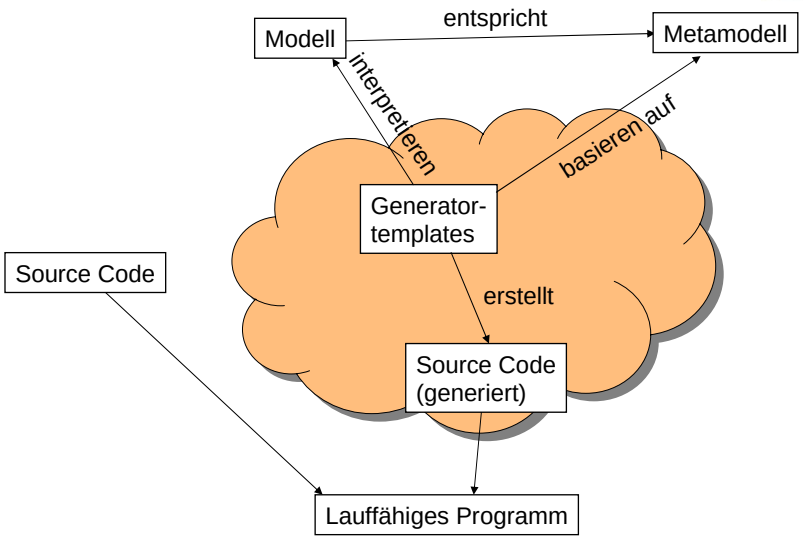
Übersicht



Generieren: Vorgehensweise

- Erstellen einer **nicht-generischen Implementierung**
  - Auch genannt Referenzimplementierung
- **Abstraktion** der Implementierung
  - Definition des **Metamodells**
  - Erstellung von Generator-Templates zur Generierung des Codes aus dem Modell
- In der Praxis zusätzlich:
  - Herausfaktorisierung der allen Instanzen **gemeinsamen Bestandteile** in Oberklassen und Bibliotheken
  - Generierung von **Hooks**, um manuell zu schreibenden Code zu integrieren

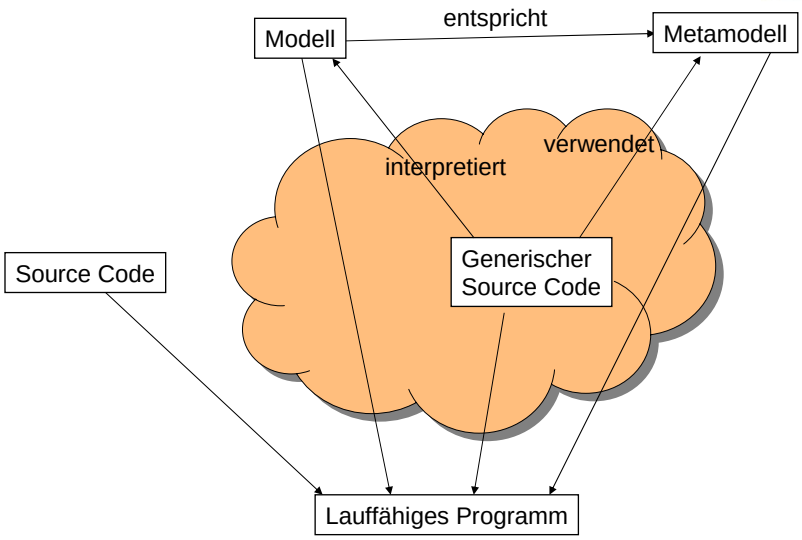
Generieren: Übersicht



Interpretieren: Vorgehensweise

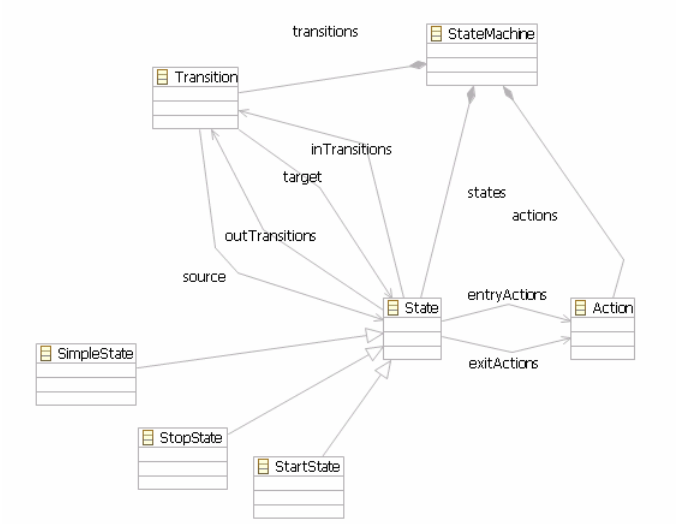
- Erstellen einer **nicht-generischen Implementierung**
- **Abstraktion** der Implementierung
  - Definition des **Metamodells**
  - Gemeinsame Bestandteile in parametrisierte Klassen und Methoden => **Generischer Source Code**
- Erstellung der **Grundkomponente** zum Laden des Modells
- In der Praxis zusätzlich: *Generierung* von
  - Basisklassen für Hooks und Callbacks
  - Zugriffsklassen auf das Modell und Metamodell

Interpretieren: Übersicht



Statemachine: Gemeinsam

Metamodell



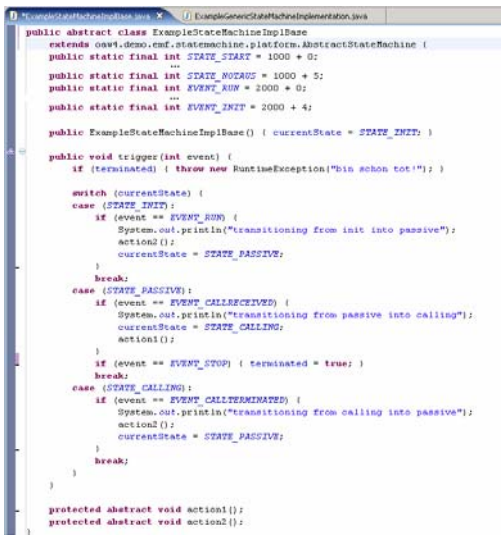
Statemachine [Generiert] (1)

Generator Templates

- Der **blaue Text** wird direkt in die Zieldatei generiert.
- Die **lila großgeschriebenen Wörter** sind Keywords der xPand Template-Sprache
- **Schwarzer Text** sind Modell-Zugriffe
- DEFINE…END-DEFINE Blöcke werden **Templates** genannt

Statemachine [Generiert] (2)

Generierter Code



```
public abstract class ExampleStateMachineImplBase
    extends osgi.demo.cmf.statemachine.platform.AbstractStateMachine {
    public static final int STATE_START = 1000 + 0;
    ...
    public static final int STATE_NOTAUS = 1000 + 5;
    public static final int EVENT_RUN = 2000 + 0;
    ...
    public static final int EVENT_INIT = 2000 + 4;

    public ExampleStateMachineImplBase() { currentState = STATE_INIT; }

    public void trigger(int event) {
        if (terminated) { throw new RuntimeException("bin schon tot!"); }

        switch (currentState) {
        case (STATE_INIT):
            if (event == EVENT_RUN) {
                System.out.println("transitioning from init into passive");
                action2();
                currentState = STATE_PASSIVE;
            }
            break;
        case (STATE_PASSIVE):
            if (event == EVENT_CALLRECEIVED) {
                System.out.println("transitioning from passive into calling");
                currentState = STATE_CALLING;
                action1();
            }
            if (event == EVENT_STOP) { terminated = true; }
            break;
        case (STATE_CALLING):
            if (event == EVENT_CALLTERMINATED) {
                System.out.println("transitioning from calling into passive");
                action2();
                currentState = STATE_PASSIVE;
            }
            break;
        }
    }

    protected abstract void action1();
    protected abstract void action2();
}
```

Statemachine [Generiert] (3)

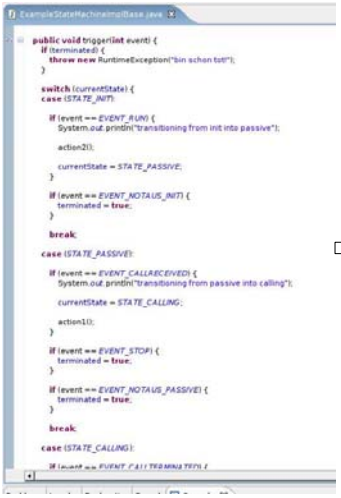
Integration manuellen Codes

ExampleStateMachineImplementation.java

```
public class ExampleStateMachineImplementation extends
        ExampleStateMachineImplBase {

    public int action1Counter;
    public int action2Counter;

    protected void action1() {
        action1Counter++;
    }

    protected void action2() {
        action2Counter++;
    }

}
```


Statemachine [Generisch] (1)

Abstraktion der Implementierung



GenericStateMachine.java

```
package oaw4.demo.emf.statemachine.platform;

import sm.Action;

public abstract class GenericStateMachine {
  protected StateMachine sm = null;
  protected State currentState = null;
  protected boolean terminated = false;

  protected GenericStateMachine(StateMachine sm) {

  public State getCurrentState() {

  public boolean isTerminated() {

  public State trigger(Transition transition) {
    if ( terminated ) throw new RuntimeException("bin schon tot!");

    if (transition.getSource() .equals(currentState)) {
      for (Object o : currentState.getExitActions()) {
        perform((Action)o);
      }
      currentState = transition.getTarget();
      for (Object o : currentState.getEntryActions()) {
        perform((Action)o);
      }
      if (currentState instanceof StopState) {
        terminated = true;
      }
    }

    return currentState;
  }

  protected abstract void perform(Action a);
}
```

Statemachine [Generisch] (2)

Generierung von Applikations-Hooks

GenericRoot.xpt

```
«IMPORT sm»
«EXTENSION templates::java»

«DEFINE Root FOR StateMachine»
  «FILE "Generic"+implBaseClassName()+".java"»
    public abstract class Generic«implBaseClassName()»
      extends oaw4.demo.emf.statemachine.platform.GenericStateMachine {

      public Generic«implBaseClassName()»(sm.StateMachine sm) {
        super(sm);
      }

      public void perform( sm.Action a ) {
        «FOREACH actions AS a»
          if ( a.getName().equals("«a.name»")) {
            «a.name»();
          }
        «ENDFOREACH»
      }

      «FOREACH actions AS a»
        protected abstract void «a.name»();
      «ENDFOREACH»

    }
  «ENDFILE»
«ENDDEFINE»
```

GenericExampleStateMachineImplBase.java

```
public abstract class GenericExampleStateMachineImplBase extends
    oaw4.demo.emf.statemachine.platform.GenericStateMachine {
  public GenericExampleStateMachineImplBase(sm.StateMachine sm) {
    super(sm);
  }

  public void perform(sm.Action a) {
    if (a.getName().equals("action1")) {
      action1();
    }

    if (a.getName().equals("action2")) {
      action2();
    }
  }

  protected abstract void action1();

  protected abstract void action2();
}
```


Statemachine [Generisch] (3)

Präsentation und Interaktion durch den Benutzer

GenericStateMachineTest.java

```
import java.io.BufferedReader;

public class GenericStateMachineTest {
  public static void main(String[] args) {
    ExampleGenericStateMachineImplementation sm =
      new ExampleGenericStateMachineImplementation(loadStateMachine());

    while(!sm.isTerminated()) {
      printOptions(sm);
      int index = readTransition();
      sm.trigger((Transition)sm.getCurrentState().getOutTransitions().get(index));
    }
    System.out.println("Your StateMachine has terminated.");
  }

  public static void printOptions(ExampleGenericStateMachineImplementation sm) {
    System.out.println("You are in State " + sm.getCurrentState().getName());
    System.out.println("action1Counter is  " + sm.action1Counter);
    System.out.println("action2Counter is  " + sm.action2Counter);
    System.out.println("Your options are:");
    for (int i=0; i<sm.getCurrentState().getOutTransitions().size(); i++) {
      System.out.println("(" + i + ") " +
          ((Transition)sm.getCurrentState().getOutTransitions().get(i)).getName());
    }
  }

  public static int readTransition() {

  private static StateMachine loadStateMachine() {

  }
```

Console | Problems | Javadoc

```
<terminated> GenericStateMachineTe
You are in State init
action1Counter is  0
action2Counter is  0
Your options are:
(0) run
>> 0
You are in State passive
action1Counter is  0
action2Counter is  1
Your options are:
(0) callReceived
(1) stop
>> 0
You are in State calling
action1Counter is  1
action2Counter is  1
Your options are:
(0) callTerminated
>> 0
You are in State passive
action1Counter is  1
action2Counter is  2
Your options are:
(0) callReceived
(1) stop
>> 0
You are in State calling
action1Counter is  2
action2Counter is  2
Your options are:
(0) callTerminated
>> 0
You are in State passive
action1Counter is  2
action2Counter is  3
Your options are:
(0) callReceived
(1) stop
>> 1
Your StateMachine has terminated.
```

Typische Anwendungsszenarien

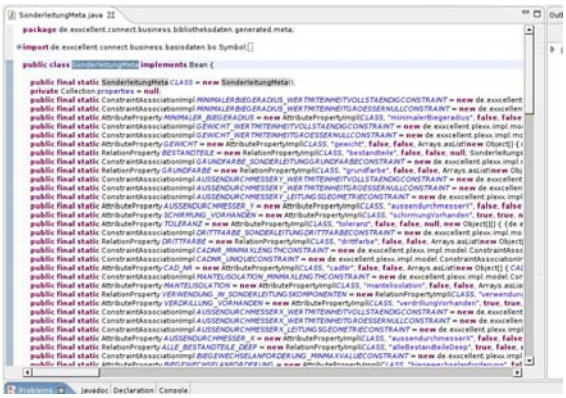
- Generierung
  - Quellcode, der bestimmten APIs genügen muss
  - Konfigurationsfiles
  - Dokumentation
  - Build Files
- Interpretiert
  - Verhalten (State Machines, Rule Engines, ...)
  - Datenvalidierung
  - Datentransformation
  - GUIs

Praxisbeispiele: Rahmenbedingungen

- Fachliches Umfeld: Verwaltungssystem für Teile aus dem Bereich Fahrzeugelektrik (Kabel, Stecker, Dichtungen, etc.)
- Technisches Umfeld:
  - Oracle Datenbank, > 100 Tabellen
  - O/R Mapping Ansatz (TopLink)
  - Rich Client (Swing) zur Dateneingabe
  - Thin Client (wingS) zur Datenrecherche
- Systemumfeld
  - Benutzer mit unterschiedlichen Sichtweisen auf die Daten
  - Datenimport/export mit mehreren Systemen
  - Datenmodell und Systemumgebung im Wandel

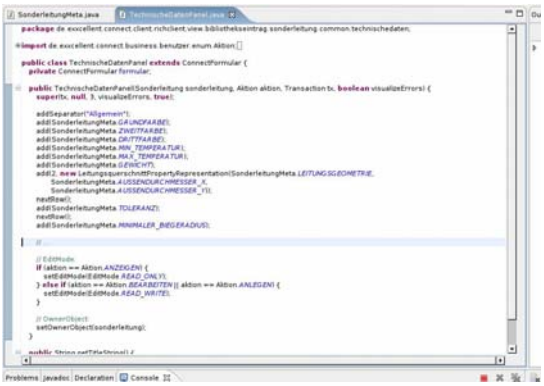
Praxisbeispiele: Modellinformationen

- Pre-EMF-Ära
- Pragmatischer Ansatz mit eigenem Metamodell
  - Klassen, Attribute, Datentypen, Beziehungen, Constraints, …
- Generiert als Konstanten

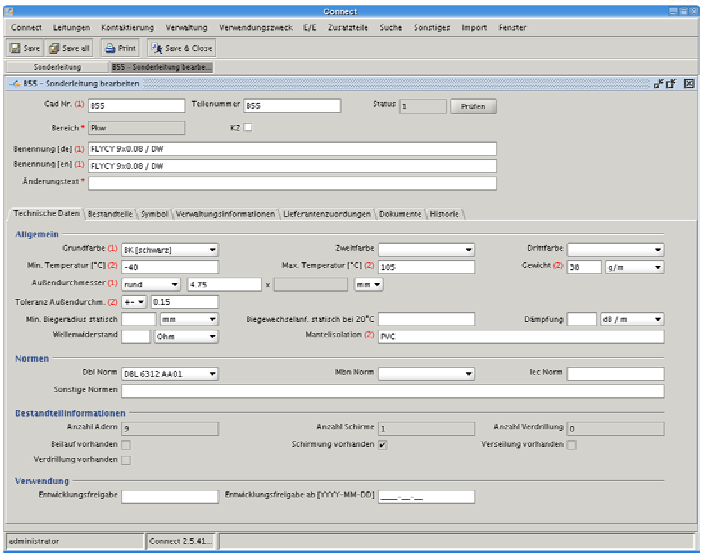


Praxisbeispiel 1: UI-Aufbau (1)

- Definition des (statischen) UIs anhand von Business-Attributen
- Keine UI-Elemente (Textfelder, etc.) im Quelltext
- Einheitliches Look&Feel
- Sehr effiziente UI Entwicklung



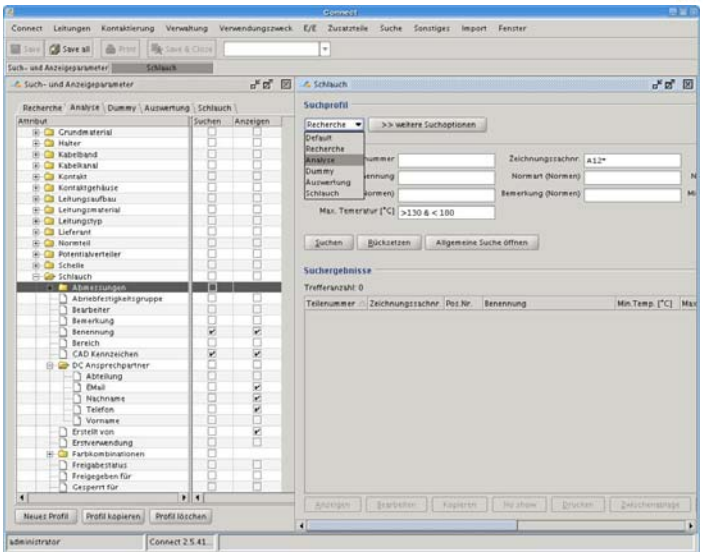
Praxisbeispiel 1: UI Aufbau (2)



Praxisbeispiel 2: Suchkonfiguration (1)

- Benutzer suchen mit verschiedenen Sichten dieselben Daten
  - Leitungssatzentwickler => technische Daten
  - Teamleiter => Statusinformationen
  - Kundendienst => Teilenummer
  - ...
- Suchprofile ermöglichen es zu definieren:
  - Nach welchen Attributen gesucht wird
  - Welche Attribute in der Treffermenge angezeigt werden
  - Vom Benutzer selbst konfigurierbar
- Technische Auswirkungen:
  - Keine fest definierten Suchdialoge
  - Keine vordefinierten Queries

Praxisbeispiel 2: Suchkonfiguration (2)



Vergleich und Bewertung (1)

	Generiert	Generisch
Flexibilität zur Laufzeit	--	++

- Generik unumgänglich für Flexibilität zur Laufzeit
  - Die Alternative wäre Code generieren, Compilieren, Dynamisch laden … nur in Ausnahmefällen praktikabel
- Reflection über das Modell möglich
- Änderungen des Verhaltens der Applikation durch Benutzerinteraktion
- Theoretisch sogar Änderung des Modells zur Laufzeit möglich!
  - Durchaus nicht unüblich, um Datenstrukturen anzupassen (User Defined Attributes)

Vergleich und Bewertung (2)

	Generiert	Generisch
Performance	+	-

- Generierter Code ist statischer Quelltext
- Generischer Code wird gesteuert durch `if` Anweisungen, Polymorhismus oder andere generische Mechanismen
- Laufzeitnachteil kann je nach Anwendungsgebiet nennenswert (embedded, oder bei großen Datenmengen in der Enterprise-Welt) bis vernachlässigbar sein
- Flexibilität für Optimierungen bei Spezialfällen auch und insbesondere beim generischen Ansatz vorsehen!

Vergleich und Bewertung (3)

	Generiert	Generisch
Entwicklungszyklen	-	+

- Generatorlauf als zusätzlicher Schritt kostet Zeit
- Größere Menge an Quelltext erfordert längere Compilezyklen
- Kaum Praxisrelevant, da
  - Auch beim generischen Ansatz meist Teile generiert werden müssen
  - Unterschiede erst bei sehr großen Projekten auftreten
- In jedem Fall lohnenswert:
  - Applikation in Teilkomponenten aufspalten, sodass bei Änderungen nur Teile generiert werden müssen

Vergleich und Bewertung (4)

	Generiert	Generisch
Applikations-Startup	+	-

- Generierter Code liegt fertig kompiliert und vollständig vor
- Generik erfordert Lesen und Auswerten des Modells
- Bei EMF basierten Ansätzen muss auch das Metamodell gelesen werden
- Bei großen Metamodellen (z.B. UML2) kann das u.U. mehrere Sekunden dauern
  - Der Bau eines Interpreters für direkte UML2 Modelle ist aber generell nicht empfehlenswert (Interpreter wird sehr kompliziert)
  - Stattdessen lieber eine Transformation von UML2 ein eigenes Metamodell, Interpreter dann darauf aufsetzen

Vergleich und Bewertung (5)

	Generiert	Generisch
Debugging	+	-

- Generierter Code ist sehr explizit
- Breakpoints lassen sich leicht und direkt setzen
- Generischer Code arbeitet auf Meta-Ebene
- Umdenken des Entwicklers erforderlich
- Breakpoints fast nur mit Conditions sinnvoll einsetzbar

Vergleich und Bewertung (6)

	Generiert	Generisch
Entwicklungseffizienz	++	++

- Beide Ansätze gleichwertig
- Wesentlich effizientere Entwicklung gegenüber manuellem Programmieren
- Weniger Fehler

Vergleich und Bewertung (7)

	Generiert	Generisch
Verifizierbarkeit	+	-

- In manchen Umgebungen muss der Quellcode bzgl. verschiedener Eigenschaften verifiziert werden:
  - Nutzung bestimmter Sprachkonstrukte
  - Deadlockfreiheit
  - Timing-/QoS Eigenschaften
- Dies ist bei generiertem Code oft einfacher, weil die algorithmische Komplexität meistens geringer ist
  - Der Interpreter enthält quasi alle Alternativen,
  - Wohingegen der generierte Code nur die enthält, die wirklich vorkommen

Vergleich und Bewertung (8)

	Generiert	Generisch
Codeumfang	+/-	+/-

- Hohe Metamodellabdeckung (große Modellen mit recht kleinem Metamodell)
  - Generisch: kleiner, da nur der Interpreter ins Gewicht fällt
  - Generiert: größer, da jedes Modellelement in Code resultiert
- Geringe Metamodellabdeckung:
  - Generisch: größer, da der Interpreter „für alle Fälle“ deployt werden muss
  - Generiert: bei kleinen Modellen kleiner, bei größeren Modellen groß
- Kommt also ganz drauf an ☺

Vergleich und Bewertung (Übersicht)

	Generiert	Generisch
Flexibilität zur Laufzeit	--	++
Performance	+	-
Entwicklungszyklen	-	+
Applikations-Startup	+	-
Debugging	+	-
Entwicklungseffizienz	++	++
Verifizierbarkeit	+	-
Codeumfang	+/-	+/-

Fazit

- Generik als alternativer Ansatz für MDSD
- In manchen Situationen sogar unumgänglich
- Vor- und Nachteile gegenüber rein generatorbasierter MDSD
- In der Praxis immer Mischformen von generiert und generisch
- Best-of-both-Worlds Ansatz adaptieren!

Aber, ist das dann noch MDSD??

Oder: was ist die **Essenz** von MDSD?

- Modelle, die bestimmte Viewpoints (Aspekte) einer Anwendung **klar, vollständig, formal und automatisch verarbeitbar** beschreiben
- Darstellung der Sachverhalte in einer **für die beteiligten Stakeholder passenden Syntax**
- **Validierung von Modellen** bzgl. bestimmter, vom Metamodell und Constraints definierter Eigenschaften überprüfen
- **Automatische Erstellung von Software** aus Modellen zwecks Konsistenz und Arbeitsersparnis
  - Entweder per Generierung
  - Oder per Interpreter