

## Generieren vs. Interpretieren

Die andere Seite von MDSD

ex|Xcellent  
solutions

**völder**  
ingenieurbüro für softwaretechnologie

### Inhalt

ex|Xcellent  
solutions

**völder**  
ingenieurbüro für softwaretechnologie

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

2

Generieren vs. Interpretieren

- *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

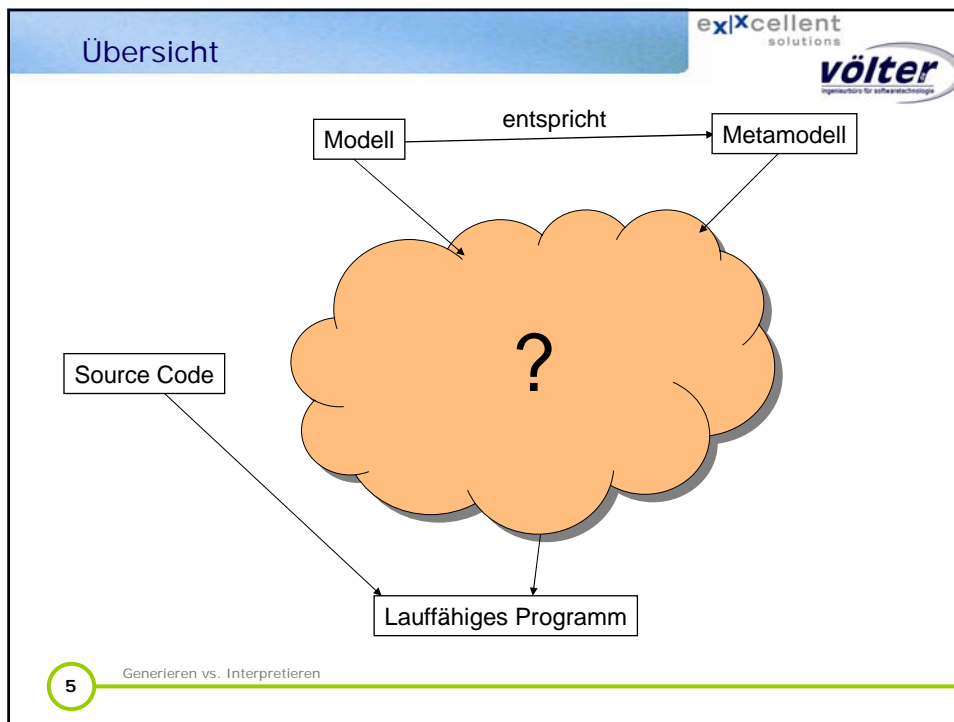
3

Generieren vs. Interpretieren

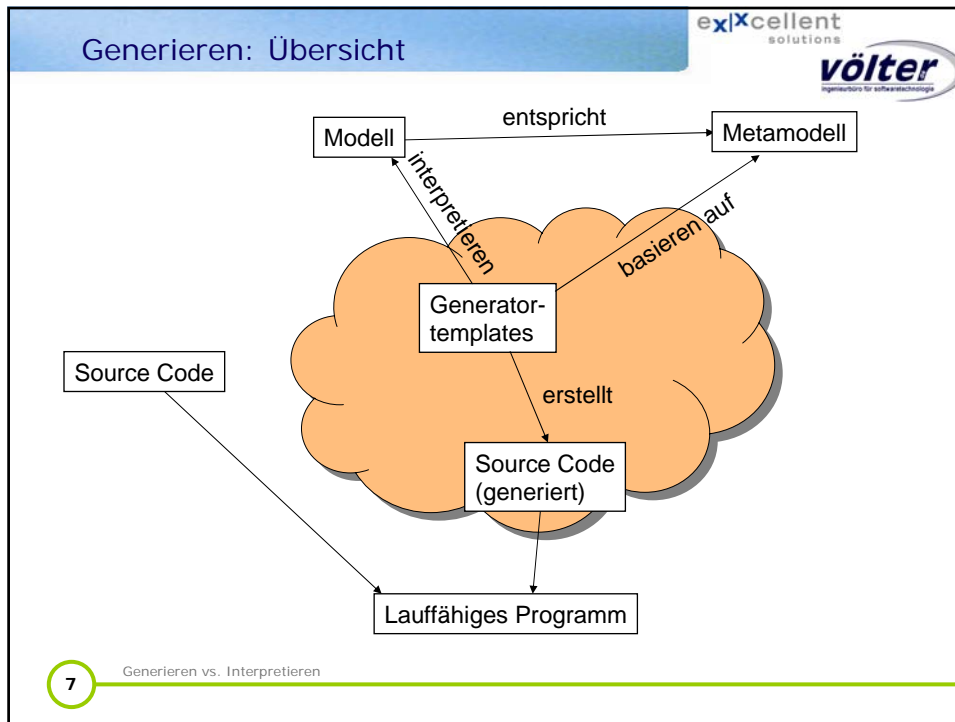
- **Abstraktion** des Problembereichs
- Definition eines **Metamodells**
- **Instantiierung** des Metamodells
- **Auswertung** des Metamodells
- Gleiches **Ziel**
  - Vermeidung wiederholter und manueller Programmierfähigkeit

4

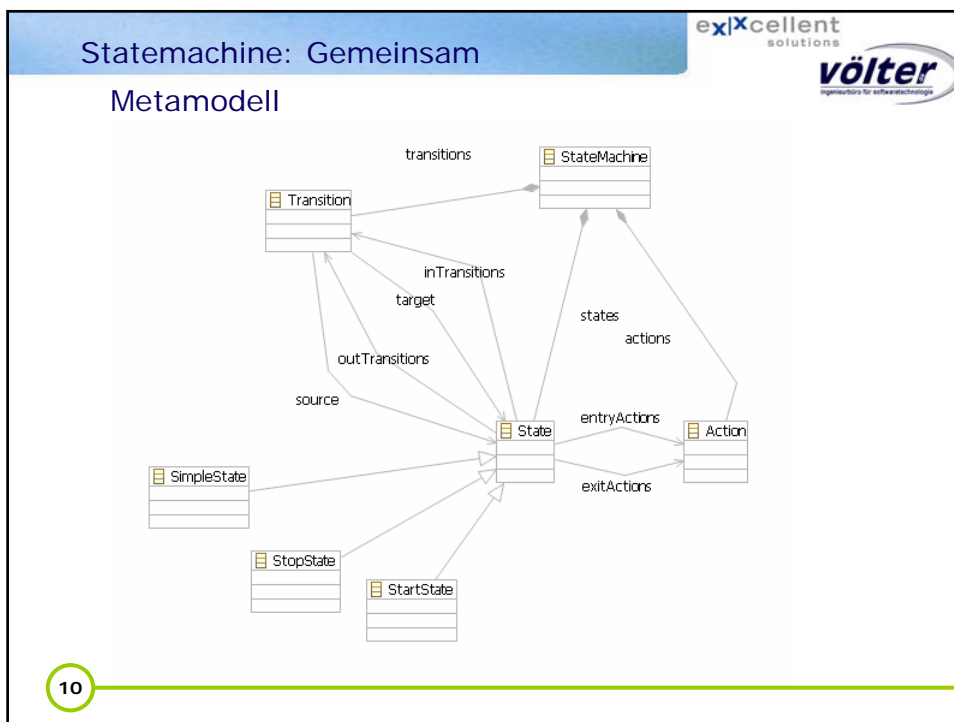
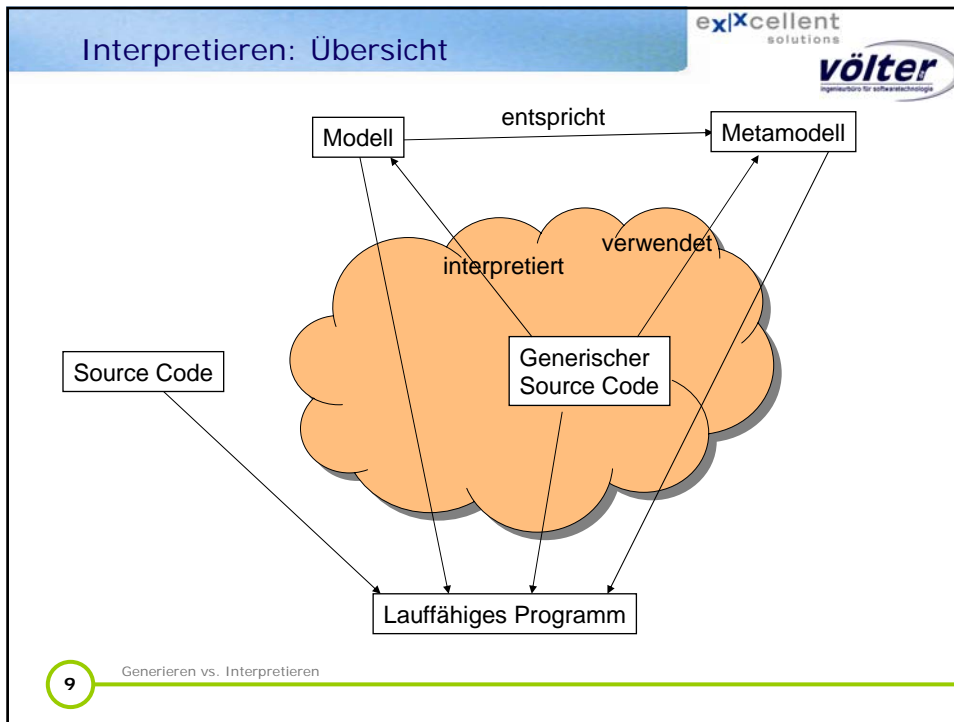
Generieren vs. Interpretieren



- Generieren: Vorgehensweise
- ex|cellent solutions  
völder  
ingenieure für softwaretechnologie
- 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
- 6 Generieren vs. Interpretieren



- Interpretieren: Vorgehensweise
- ex|cellent solutions  
völder  
ingenieure für softwaretechnologie
- 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
- 8 Generieren vs. Interpretieren



## Statemachine [Generiert] (1)

### Generator Templates

```

>IMPORT simpleSE.
>EXTENSION templates:GeneratorUtil.
>DEFINE File FOR StateMachine.
>FILE basePath()+"Abstract"+name.toFirstUpper()+"_java">
package <packageName>;

public abstract class <impl>BaseClass<> implements <actionInterfaceName> {
    private <stateEnumName>() currentState = <initialState>.stateId(this);
    private boolean terminated = false;

    public void handleEvent( <eventEnumName>() event ) {
        if ( terminated ) throw new RuntimeException( "this sm is terminated!" );
        switch ( currentState ) {
            >FOR EACH states AS s=>
            case <stateId>() =>
                >FOR EACH s.transitions AS t=>
                if ( event == t.event.eventId(this) ) {
                    >EXPAND executeTransition(this) FOR t;
                    break;
                }
                >EXPAND handleIllegalTransition.
                >ENDFOR EACH.
                break; / break out if no suitable transition has been found'
            >ENDFOR EACH.
        }
    }

    public <stateEnumName>() getCurrentState() {
        return currentState;
    }
}
<ENDFILE>.
<ENDDEFINE>.
>DEFINE handleIllegalTransition FOR StateMachine.
<ENDDEFINE>.
>DEFINE executeTransition(StateMachine sm) FOR Transition.
>FOR EACH actions AS a=>
    call a.methodName();
<ENDFOR EACH>.
currentState = <to>.stateId(sm);
<ENDDEFINE>.

```

- 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

11

Generieren vs. Interpretieren

## Statemachine [Generiert] (2)

### Generierter Code

```

public abstract class ExampleStateMachineImplBase
extends org.dcmi.emf.statemachine.platform.AbstractStateMachine {
    public static final int STATE_START = 1000 + 0;
    public static final int STATE_NORMAL = 1000 + 5;
    public static final int EVENT_RUN = 2000 + 0;
    public static final int EVENT_ZENT = 2000 + 4;

    public ExampleStateMachineImplBase() { currentState = STATE_ZENT; }

    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");
                    action1();
                    currentState = STATE_PASSIVE;
                }
                break;
            case (STATE_PASSIVE) :
                if ( event == EVENT_CALLERCEIVED ) {
                    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();
}

```

12

Generieren vs. Interpretieren

## Statemachine [Generiert] (3)

### Integration manuellen Codes

```
ExampleStateMachineImplementation.java X
public class ExampleStateMachineImplementation extends
    ExampleStateMachineImplBase {

    public int action1Counter;
    public int action2Counter;

    protected void action1() {
        action1Counter++;
    }

    protected void action2() {
        action2Counter++;
    }
}
```

13

Generieren vs. Interpretieren

## Statemachine [Generisch] (1)

### Abstraktion der Implementierung

```
ExampleStateMachineImplBase.java X
public void triggerListEvent() {
    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;
            }
            if (event == EVENT_NOTAUS_INIT) {
                terminated = true;
            }
            break;
        case (STATE_PASSIVE):
            if (event == EVENT_CALLERCEIVED) {
                System.out.println("transitioning from passive into calling");
                currentState = STATE_CALLING;
                action3();
            }
            if (event == EVENT_STOP) {
                terminated = true;
            }
            if (event == EVENT_NOTAUS_PASSIVE) {
                terminated = true;
            }
            break;
        case (STATE_CALLING):
            if (event == EVENT_CALL_TERMINA_TEPIL)
    }
```



```
GenericStateMachine.java X
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);
}
```

14

Generieren vs. Interpretieren

## Statemachine [Generisch] (2)

ex|cellent  
solutions

völder  
ingenieure für softwaretechnologie

### 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»
«ENDEDEFINE»

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();
}
```

15

Generieren vs. Interpretieren

## Statemachine [Generisch] (3)

ex|cellent  
solutions

völder  
ingenieure für softwaretechnologie

### 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
>> 1
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
>> 1
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.
```

16

Generieren vs. Interpretieren

## Typische Anwendungsszenarien

ex|Xcellent  
solutions



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

17

Generieren vs. Interpretieren

## Praxisbeispiele: Rahmenbedingungen

ex|Xcellent  
solutions



- 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

18

Generieren vs. Interpretieren



## Praxisbeispiel 1: UI Aufbau (2)

ex|cellent  
solutions

völder  
ingenieure für softwaretechnologie

The screenshot shows a software window titled "BSS - Sonderleitung bearbeiten". It contains several sections for data entry:

- Header:** CAD Nr. (155), Teilnummer (155), Status (1), and a "Drucken" button.
- General Data:** Bereich (Pflanz), KZ (checkbox), Benennung (de) (HLVCT 9h0.08 / DW), Benennung (en) (HLVCT 9h0.08 / DW), and Änderungszeit.
- Technical Data:** Grundfarbe (1) (BK [schwarz]), Zweifarbe, Drittfarbe, Min. Temperatur (°C) (2) (-40), Max. Temperatur (°C) (2) (105), Gewicht (2) (30 g/m), Außendurchmesser (1) (rund), 4.75 x mm, Toleranz Außendurchm. (2) (+/- 0.15), Min. Biegedruck statisch (mm), Biegeweichleitf. statisch bei 20°C, Dämpfung (dB / m), Wellenlängenstand (Cable), Mantelisolierung (2) (NYC).
- Standards:** Deut Norm (DIN 6312 AA61), Min Norm, Iec Norm, and Sonstige Normen.
- Inventory Information:** Anzahl Adern (g), Anzahl Schirme (1), Anzahl Verdrillung (0), Teil auf vorhanden (checkbox), Schirmung vorhanden (checkbox checked), Verdrillung vorhanden (checkbox).
- Viewing:** Entwicklungs freigabe, Entwicklungs freigabe ab (YYYY-MM-DD).

21

Generieren vs. Interpretieren

## Praxisbeispiel 2: Suchkonfiguration (1)

ex|cellent  
solutions

völder  
ingenieure für softwaretechnologie

- 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

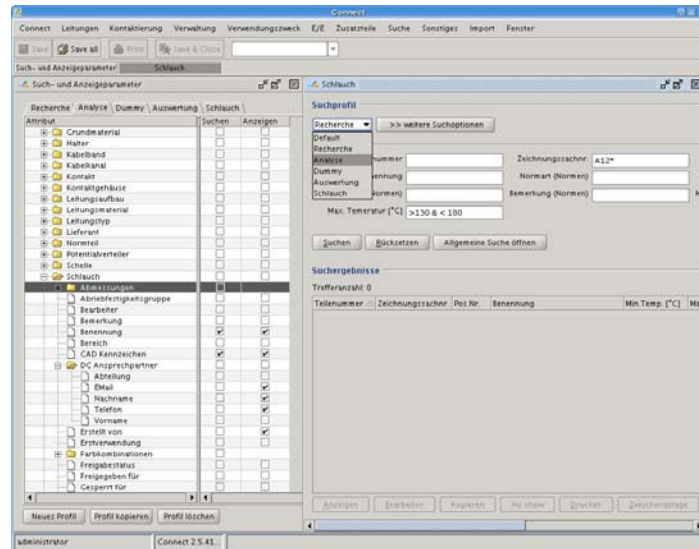
22

Generieren vs. Interpretieren

## Praxisbeispiel 2: Suchkonfiguration (2)

excellent  
solutions

**völder**  
ingenieure für softwaretechnologie



23

Generieren vs. Interpretieren

## Vergleich und Bewertung (1)

excellent  
solutions

**völder**  
ingenieure für softwaretechnologie

	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)

24

Generieren vs. Interpretieren

## Vergleich und Bewertung (2)

ex|Xcellent  
solutions



	Generiert	Generisch
Performance	+	-

- Generierter Code ist statischer Quelltext
- Generischer Code wird gesteuert durch `if` Anweisungen, Polymorphismus 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!

25

Generieren vs. Interpretieren

## Vergleich und Bewertung (3)

ex|Xcellent  
solutions



	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

26

Generieren vs. Interpretieren

## Vergleich und Bewertung (4)

ex|Xcellent  
solutions



	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

27

Generieren vs. Interpretieren

## Vergleich und Bewertung (5)

ex|Xcellent  
solutions



	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

28

Generieren vs. Interpretieren

## Vergleich und Bewertung (6)

ex|Xcellent  
solutions



	Generiert	Generisch
Entwicklungseffizienz	++	++

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

29

Generieren vs. Interpretieren

## Vergleich und Bewertung (7)

ex|Xcellent  
solutions



	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

30

Generieren vs. Interpretieren

## 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 😊

31

Generieren vs. Interpretieren

## Vergleich und Bewertung (Übersicht)

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

32

Generieren vs. Interpretieren

- 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!

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 Arbeitersparnis
  - Entweder per Generierung
  - Oder per Interpretierer