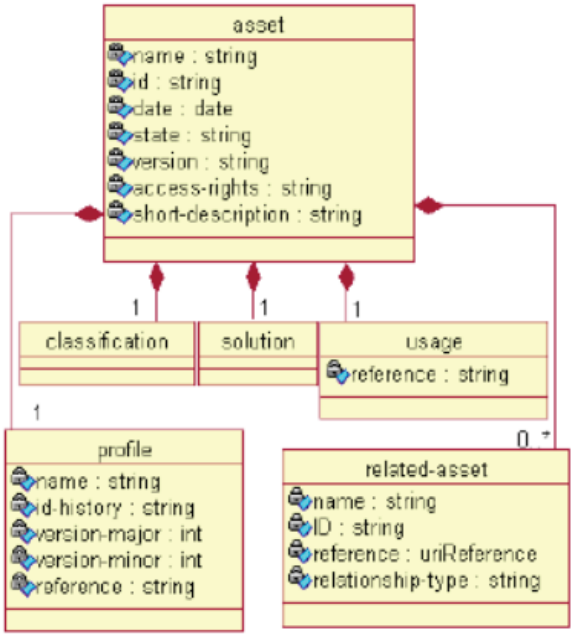
9. Wiederverwendung



Dr.-Ing. Clemens Reichmann

Clemens.Reichmann@vector.com,
Tel. 0721-91430-200

Institut für Technik der Informationsverarbeitung
Fakultät für Elektrotechnik & Informationstechnik
Universität Karlsruhe (TH)

Inhalt 9. Implementierung

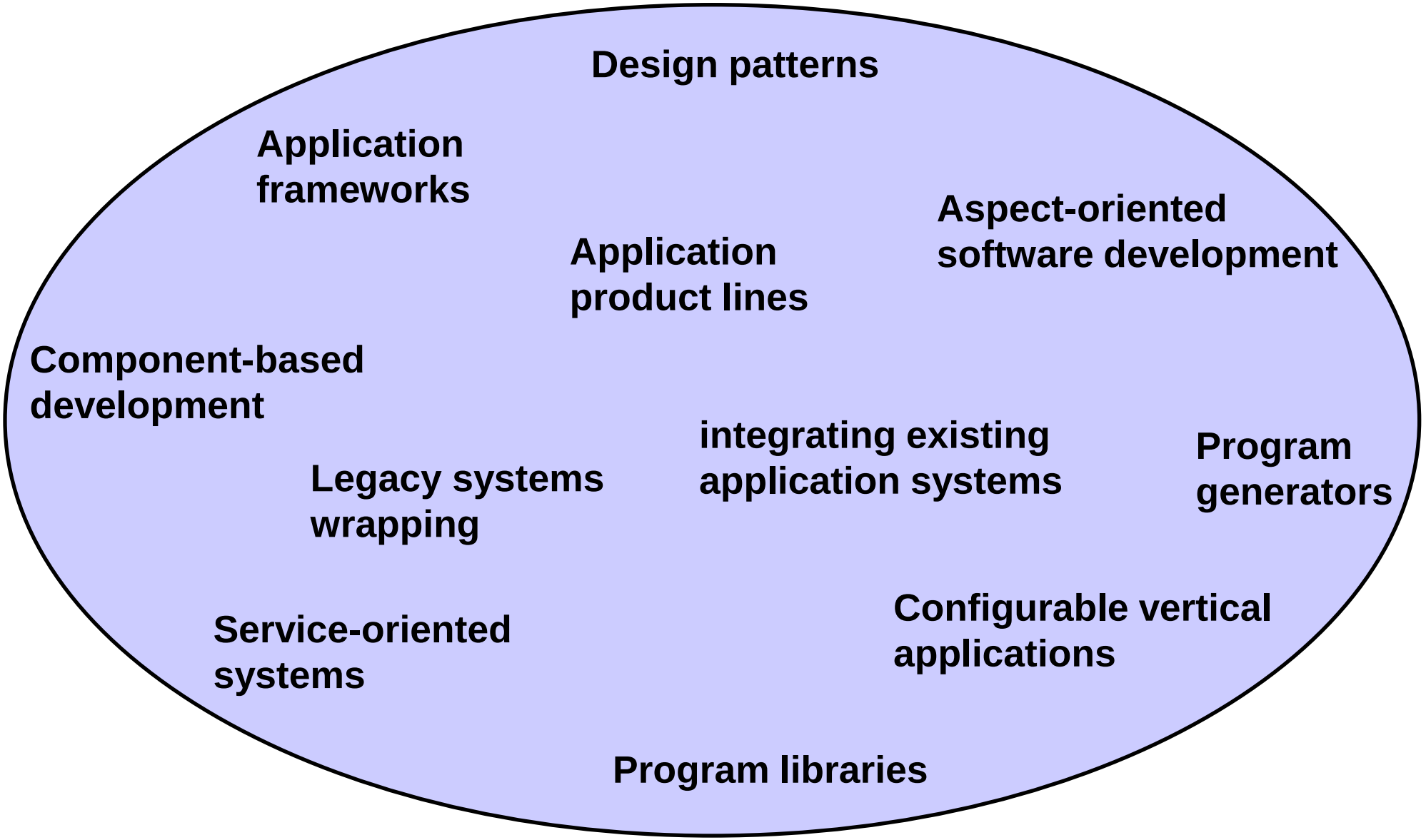
Literatur

- H. Mili, A. Mili, S. Yacoub, E. Addy, "Reuse-based Software Engineering", John Wiley & Sons, Inc., 2002
- I. Jacobson, M. Griss, P. Jonsson, "Software Reuse: Architecture, Process and Organization for Business Success," ACM Press, 1997.
- J. Sametinger, "Software Engineering with Reusable Components," Springer-Verlag, ISBN 3-540-62695-6, 1997.
- E. Karlsson, "Software Reuse: A Holistic Approach," John Wiley and Sons Ltd, 1995.
- W. Schaefer, R. Prieto-Diaz, M. Matsumoto, "Software Reusability", Ellis Horwood, 1994.
- T.J. Biggerstaff, A.J. Perlis, "Software Reusability: Volume I Concepts and Models," ACM Press, 1989.
- T.J. Biggerstaff, A.J. Perlis, "Software Reusability: Volume II Applications and Experience," ACM Press, 1989.

9.1 Warum Software-Reuse ?

- Verkürzung der Entwicklungszeit
- Erhöhung der Produktivität
- Verbesserung der Vorhersagbarkeit (Kosten)
- Verbesserung der Qualität
- Management zunehmender Komplexität
- Vereinfachung der Wartung

Umfeld der Wiederverwendung



Das „NIH-Syndrom“

- Erscheinungsformen des **„Not-Invented-Here“**-Syndroms:

„Rad immer wieder neu erfinden.“

- „Bevor ich etwas Wiederverwendbares gefunden und angepasst habe, habe ich es dreimal selber geschrieben.“
- „Wenn ich Code wiederverwende, sinkt meine Leistung (gemessen in **L**ines-**O**f-**C**ode / Zeit).“
- „Ich vertraue nur in Code, den ich selbst geschrieben habe.“
- „Fremden Code zu verwenden ist geistiger Diebstahl.“

→ **Ist was dran an diesen Argumenten?**

Warum das „NIH-Syndrom“ schädlich ist

- Der Wert von **Wiederverwendung** liegt weniger in der zeitlichen Ersparnis als in der **Qualitätsverbesserung**: Wiederverwendeter Code wird ***besser getestet und dokumentiert*** und davon profitieren alle.
- Ziel der Software-Entwicklung ist *nicht* möglichst umfangreicher Code: E. W. Dijkstra spricht vom ***„Verbrauchen von LOCs“***. Mit dem Umfang von Code steigt der **Wartungsaufwand**.
- Echter **Fortschritt** in der Software-Entwicklung ist nur möglich, wenn **arbeitsteilig** gearbeitet wird! Jeder Entwickler arbeitet mit Werkzeugen, die andere bereitgestellt haben.

Herkömmliche Wiederverwendung

- Code-Wiederverwendung mit ***„Copy-Paste-Modify“***
  - ***einfach und flexibel*** einsetzbar => populär
  - Vorteile „echter“ Wiederverwendung nicht genutzt: mit Umfang wächst das ***Wartungsproblem***,
- ***Qualitätsverbesserung*** höchstens lokal
  - ***Generische Einheiten*** (Ada, C++, Java)
  - „geplante“ Wiederverwendung => ***einfach und sicher***
  - ***unflexibel*** in Bezug auf neue Anforderungen
  - geeignet für ***ausgereifte, standardisierte*** Einheiten

Objektorientierung und Wiederverwendung …

- Die Versprechen: Objektorientierung
  - Objektorientierung...
    - ...überbrückt die Analyse-Design-Lücke
    - ...erhöht Wiederverwendung
  - Objektorientierte Systeme sind...
    - ...einfach erstellbar
    - ...qualitativ besser
- Die Realität
  - höhere Qualifikation der Entwickler erforderlich
  - Trennung von Analyse- und Design-Modell(?)
  - kaum Wiederverwendung
  - qualitativ niedrige objektorientierte Altlasten

Die Fehler

- Sprache und Denkweise wird oft gleichgesetzt
- Ausbildung der Entwickler wird vernachlässigt
- Voreiliger Analyse-Design-Wechsel
- Unzureichende Spezifikationen
- Objektorientierung = Vererbung?
- Falsche Verwendung von Vererbung/Polymorphie
- Hierarchien erst tief, dann flach, dann was?

Vererbung und Polymorphie

- Code-Reuse vs. Subtyping
- Konzeptionelle Vererbung vs. Implementierung
- Fehlerhaftes Subtyping verhindert Wiederverwendung
  - Ersetzbarkeit (Liskov Principle of Substitutability)
  - Spezialisierungen

Konzeptionelle Vererbung ...und in Programmiersprachen

Verfeinerung

Spezialisierung

Erweiterung

Code-Reuse

Implementierung

Subtyping

Code-Reuse

Spezialisierungsproblematik

- Spezialisierungen sind Bestandteil fast jedes Modells
  - Effizienz
  - Präzision
  - Einfachheit
- Beispiel:
  - „Ist ein Quadrat ein Rechteck?“
  - „Wer allgemeine Versicherungsbedingungen kennt, der kennt auch die speziellen?!“

Problematik am Beispiel JDK 1.1

- Ein **FileDialog** ist ein **Dialog**
- Ein **Dialog** ist ein **Window**
- Ein **Window** ist ein **Container**
- Zu einem **Container** können beliebige(?) **Components** hinzugefügt werden!

```
FileDialog fd = new FileDialog();
fd.addComponent(new Button(„Press me!“));
```

Auswirkungen von Spezialisierungen

- Phänomene:
  - Eingeschränkte Verwendbarkeit einer Unterklasse
  - Bisher korrekte Systeme werden plötzlich fehlerhaft
  - Einführung einer speziellen Klasse kann Überarbeitung aller Oberklassen erfordern!

- Ursache:
  „Im allgemeinen kann ich ..., aber nicht, wenn ...!“

Hellseher-Problematik

Der perfekte Entwickler (einer Klasse)...

...kennt alle zukünftigen Unterklassen

...sorgt für polymorphe Verwendbarkeit

...existiert nicht

$$\textbf{Zeitdruck} \sim \frac{1}{\textbf{hellseherische Fähigkeiten}}$$

Wege aus der „Krise“?

- Kleine (abstrakte) Schnittstellen, weiche Semantik?
- Einfache, verständliche, nicht komplexe Einheiten können wieder verwendet werden
- Vernetzungsgrad der Software darf nicht hoch sein
- Prozesse sind sehr viel schwieriger wieder zu verwenden als Funktionsbibliotheken
- Präzise Formulierung der Semantik
- Dokumentation
  - Schnittstellen (+ implizite Annahmen)
  - Tutorials, Beispiele, Motivation
  - Sourcecode erklärt innerer Aufbau → JAVA decompilieren
- Entwicklerdisziplin:
  - Spezifikation vor Implementierung
  - Dokumentation vor Implementierung(?)

Reuse - State of the Art

- Wiederverwendung bedeutet
  - Anpassung
  - Konfiguration
- Wiederverwendung durch Delegation nicht Vererbung
  - Einfacher austauschbare Teilfunktionalität
- Implementierung ist der Konfiguration vorzuziehen
  - Z.B. Template Factory, Template Pattern
  - Wegen Fehlererkennung zur Compilezeit
- Wiederverwendung kommerzieller Komponenten
- Wiederverwendung von Konzepten

  → Entwurfsmuster
- Kaum Wiederverwendung von eigenem Code!

Wiederverwendung = Flexibilität?

- Flexibilitätswahn!
- Flexibilität...
  - ...kostet Entwicklungsaufwand
  - ...garantiert keine Wiederverwendung
  - ...garantiert keine bessere Wartung / Erweiterung
- Entwicklung flexibler Komponenten geht daneben!
- Besser mehr Dokumentation, als zuviel Flexibilität?!

Wiederverwendung

- Wiederverwendung =
  Flexibilität + Robustheit + Effizienz + Klarheit
- Wiederverwendet werden Teile, ...
  - ...die bekannt sind
  - ...die zuverlässig sind
  - ...die sich bewährt haben
  - ...die besser sind als Hausgemachtes
  - ...mit denen Entwickler umgehen können
  - ...die sich in das aktuelle Umfeld eingliedern lassen

→ Bei der Wiederverwendung müssen Änderungen durchgeführt werden!!!!

Erhöhung des Reuse-Potentials

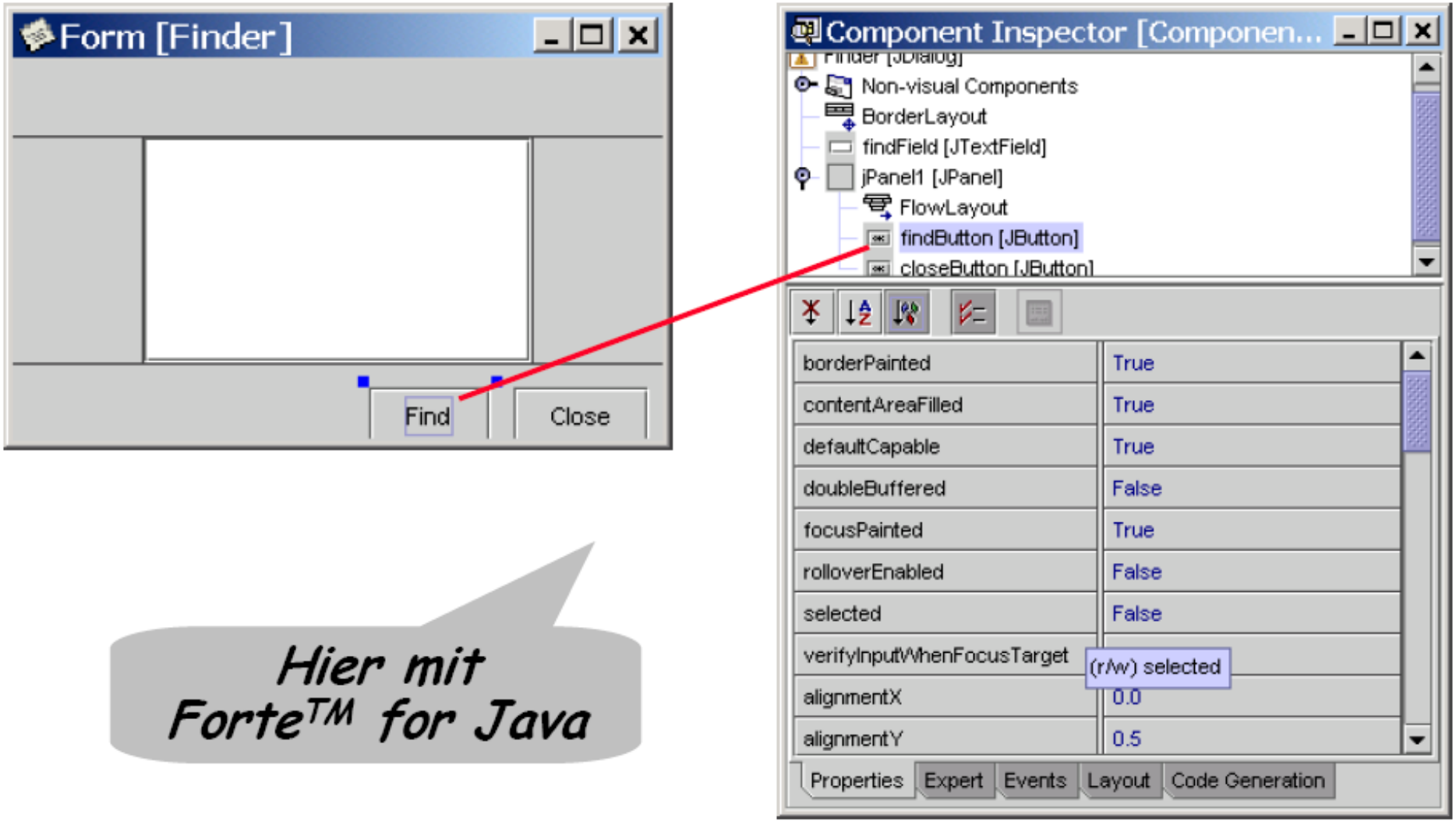
- Klassenbibliothek / Funktionenbibliothek
- Einsatz bewährter Architekturen (Dokumentation)
  - Entwurfsmuster
  - Frameworks
- Komponenten
- Reuse-Management
  - Recherche
  - Dokumentation
- Eigener Code:
  - Klassen versus Interfaces

Fragen ?

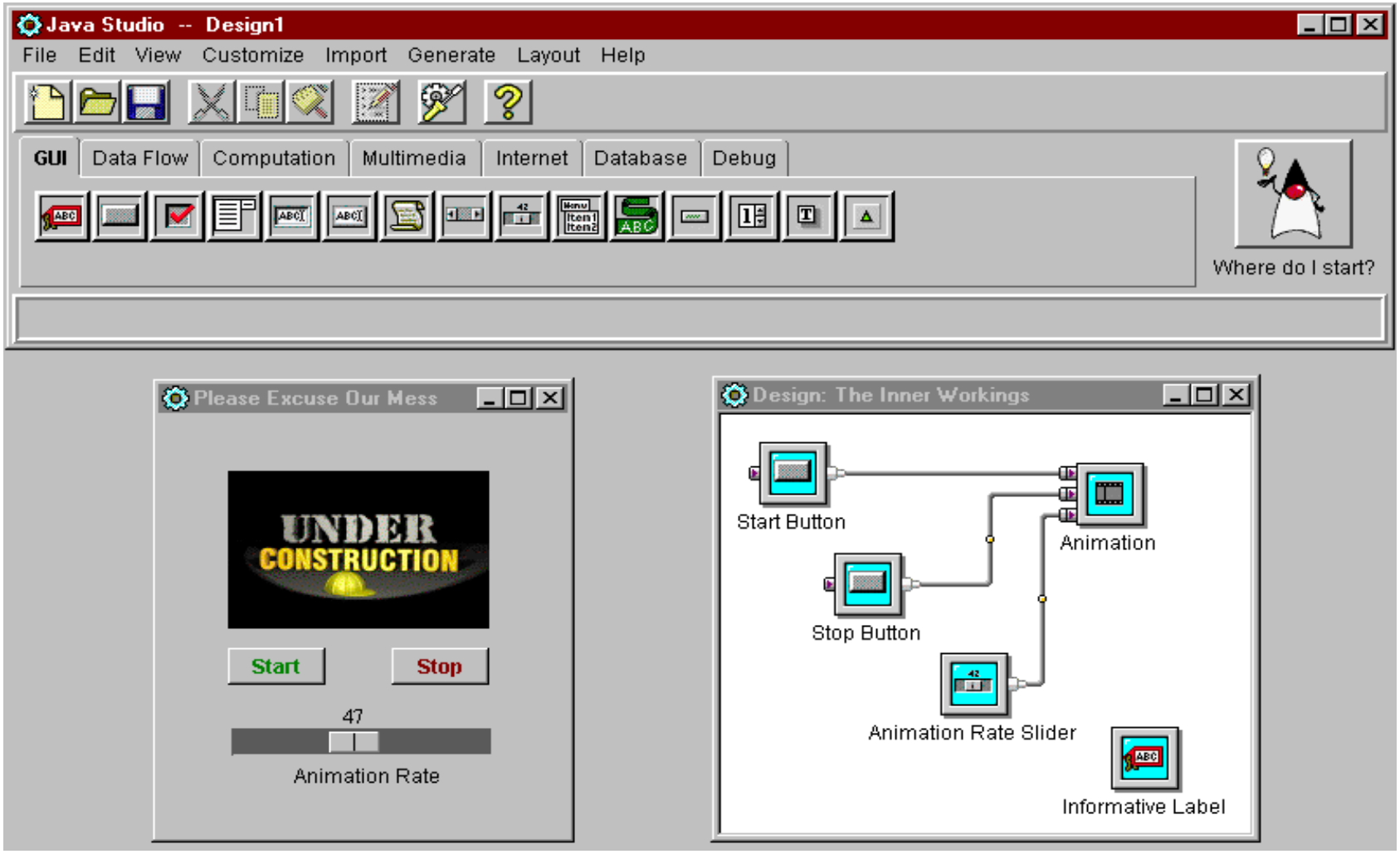
- Existierende Varianten
  - Delphi
  - JavaBeans
  - Active-X Controls
- Erweiterung des Klassenbegriffs
  - Eigenschaften
  - Methoden
  - Kopplung (Ereignisse, Pipe-Filter, ...)
- ***Anpassung ausgereifter Komponenten*** (z.B. **Java Beans**) alternativ durch Ausfüllen von ***Eigenschaftsformularen*** („Property sheet“), basierend auf getter-/setter
  => typisch bei Oberflächenelementen

In IDE in Verbindung mit „Drag&Drop“-Editoren

z.B. vorbereitete GUI-Komponente einpassen:



Komponentenbasierte Entwicklung



Komponentensysteme

Vorteile:

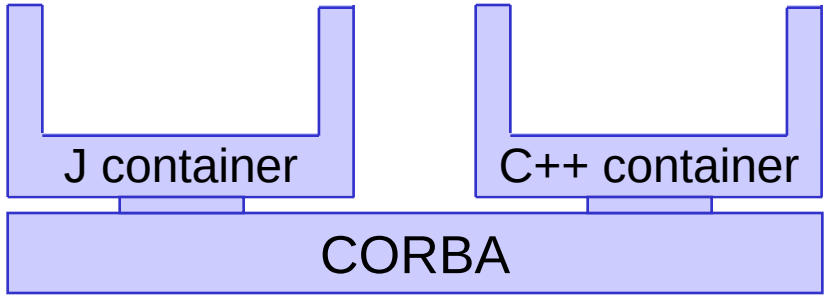
- bieten Adaption existierender Teilsysteme
- suggerieren Austauschbarkeit einzelner Komponenten

Nachteile:

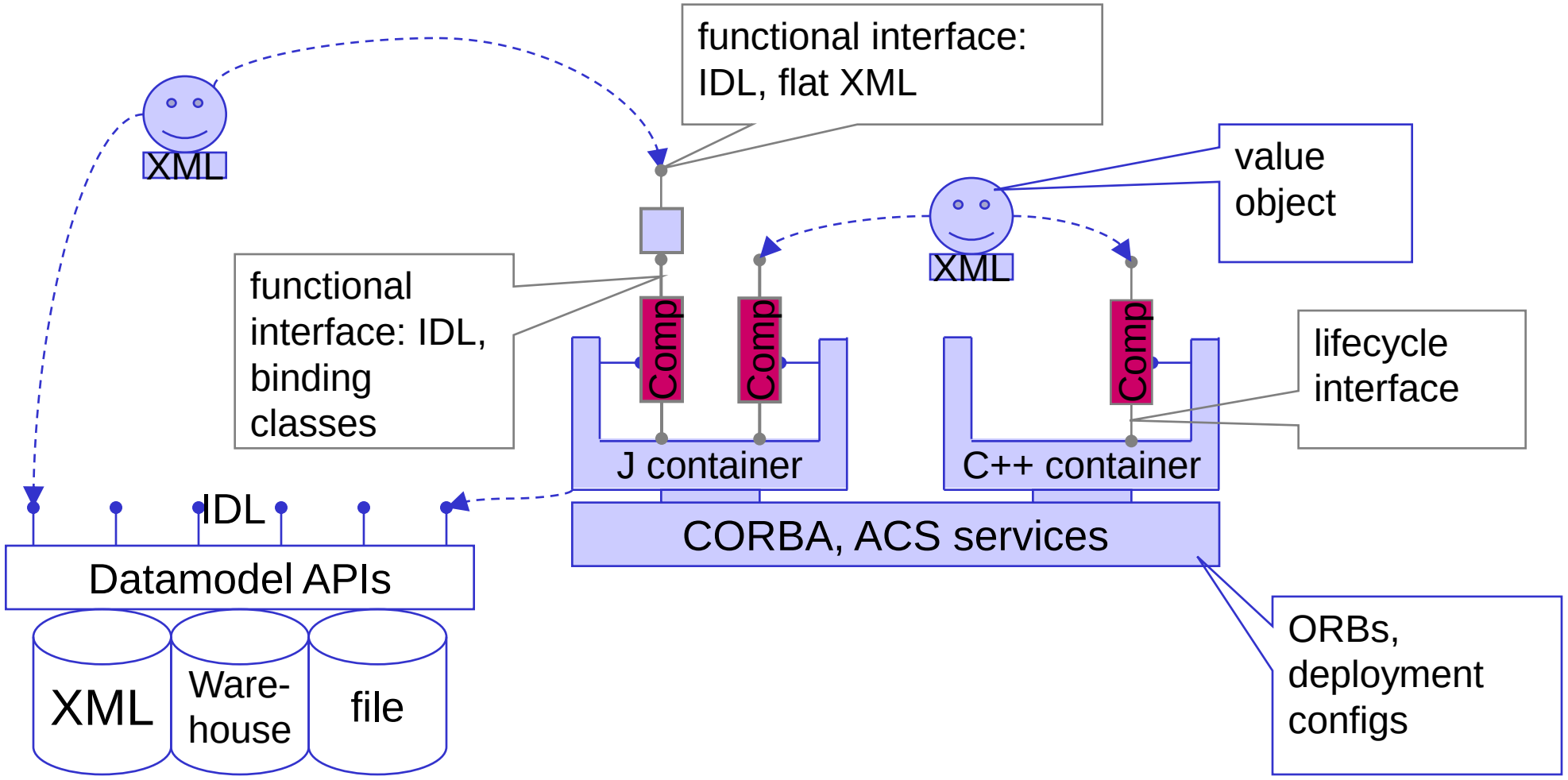
- Kompatibilitätsproblematik (passende Versionen)
- Suchproblematik (vgl. Reuse-Management)
- Nur verwendbar wenn keine Anpassungen nötig sind, die nicht vorhergesehen waren

Trends

- Middleware-Technologie (DCOM, CORBA)
- Architektursysteme (Unicon, Darwin, Rapid, AESOP)
- Meta-Programming (Anpassen existierender Teile)
- Sprachen rücken in den Hintergrund
- Bibliotheken rücken in den Vordergrund



Architektur Diagramm



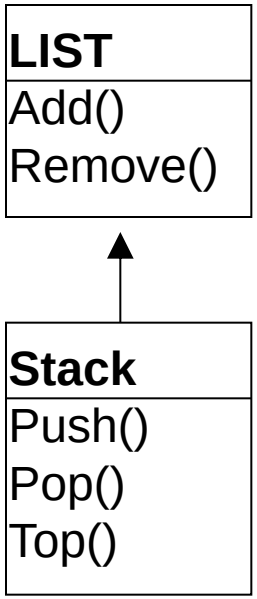
Fragen ?

9.2 Wiederverwendung: Klassen vs. Interface-Vererbung

- Interface-Vererbung:
  - In Java realisiert beispielsweise als Vererbung von abstrakten Klassen
  - Es werden nur Interfaces ohne Implementierung vererbt
- Klassen-Vererbung:
  - Konkrete Implementierung der Interfaces wird mitvererbt

Vererbung vs. Komposition

Bsp.: eine bereits implementierte Listen-Klasse soll um eine Stack-Klasse erweitert werden. Hier durch **Vererbung:**

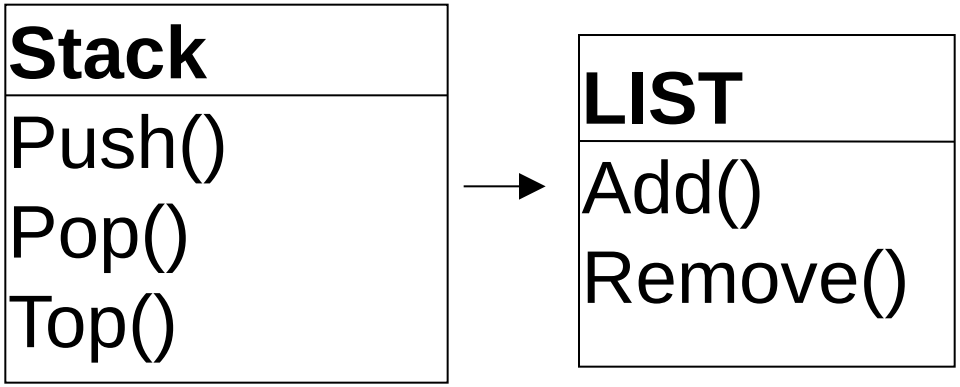


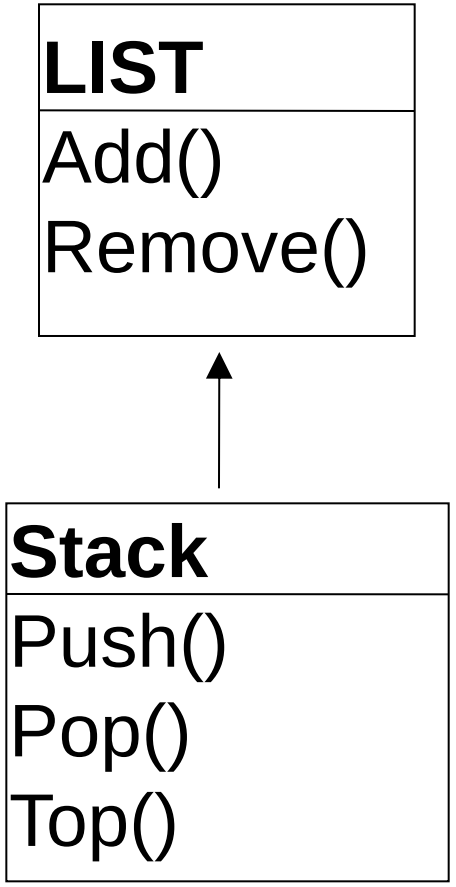
Problem:

Einige Vererbte Operationen zeigen unerwartetes Verhalten.

Was passiert, wenn der Stack-User statt Pop() Operation Remove() aufruft?

Komposition statt Vererbung





Man spricht bei Komposition von „**Blackbox-reuse**“, bei der die Unterklasse keine Kenntnis über die Interna der Oberklasse hat.

Im Gegensatz dazu wird bei Vererbung von „**Whitebox-reuse**“ gesprochen

Vererbung vs. Komposition

- Vererbung
  - Einfach zu benutzen, da von Programmiersprache direkt unterstützt
  - Implementierung kann zur Laufzeit nicht verändert werden
  - Änderungen in der Oberklasse verändern automatisch die Unterklassen → VORSICHT!
  - Zerstört Datenkapselung

Vererbung vs. Komposition

- Komposition
  - Schwerer zu implementieren
  - Datenkapselung bleibt erhalten, da Objekte nur durch Interfaces angesprochen werden
  - Weniger Abhängigkeiten
  - Klassenhierarchie bleibt überschaubar

Fragen ?

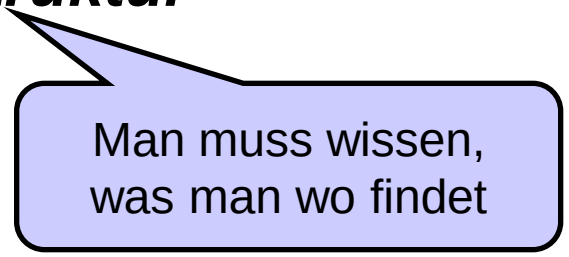
- Seit Smalltalk: **umfangreiche Klassenbibliothek** als **Plattform für Software-Entwickler**.
  - viele ***„direkt verwendbare“*** Komponenten
  - ***flexible Anpassung und Erweiterung*** möglich durch Vererbungsmechanismus
- Sehr gute Wiederverwendung möglich
- Kleine, überschaubare, unabhängige Einheiten werden wieder verwendet
- Klare abgeschlossenen Funktionen!
- Problem: API muss stabil bleiben → schwierige Weiterentwicklung der Klassenbibliothek

Java-Klassenbibliothek (1)

- wächst rasant => ***codifiziertes Informatik-Wissen***
  - **java:** Datenstrukturen, Zeichenkettenverarbeitung, Sortier- und Suchalgorithmen, Ströme, Sockets & RMI, Sicherheit, Applets, ...
  - **javax:** „Java Foundation Classes“ mit **Swing**, **Grafik**, ...
  - Spezialpakete für **XML**, **JDBC**, ...
  - **Java Enterprise Beans:** Infrastruktur für verteilte kommerzielle Anwendungen
- wegen des (ansteigenden) Umfangs selbst für Spezialisten *kaum vollständig zu „beherrschen“*
- aber: wertvolle ***Fundgrube für eigene Entwicklung***

Java-Klassenbibliothek (2)

- wichtig: ***zentral „moderiertes“ Repository*** " ***vermeidet Mehrfachentwicklung, klare Struktur***



- Nutzer profitieren von der ***Kompetenz der Spezialisten***, wie für Datenstrukturen und Algorithmen angedeutet wurde.
- ***„Offenes“ System***: Quellcode von jedermann einsehbar.
  - ***Nützlich zum Lernen***
  - ***Basis für eigene Entwicklungen / Verbesserungen***
- Einheitliche Form der ***Dokumentation*** mit Tool **javadoc,** das allen Programmierern zur Verfügung steht.
  - ***Standards erleichtern Zusammenarbeit***

Fragen ?

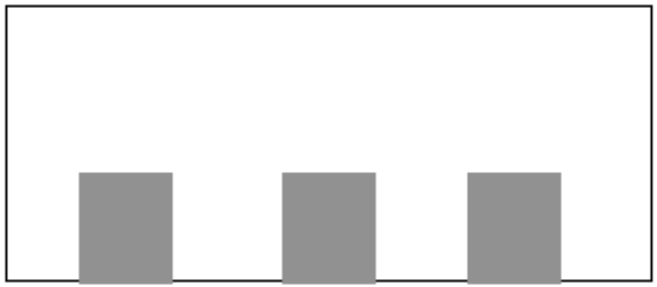
- Im Unterschied zu Klassenbibliotheken und Rahmenwerken enthalten Muster ***keine Software***.
- Muster beschreiben ***Erfahrungen***, nach denen Software erstellt werden kann.

→ siehe Kapitel 6

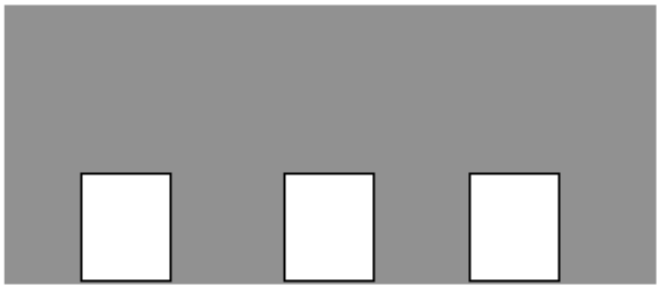
- **Definition:**
  - Ein ***„Framework“*** (Rahmenwerk, Anwendungsgerüst) ist eine ***„Halbfertiglösung“*** für eine Klasse von Problemen.
- „Technisch“ besteht ein Framework aus einer **Menge von Klassen**, die in ihrer Funktionalität und in ihrem Zusammenspiel einen **abstrakten Entwurf für die Problemfamilie** darstellen.
- Folgende Ziele sind charakteristisch:
  - Wiederverwendung von Code ***und*** von Entwurfsprinzipien
  - Wiederverwendung des ***Kooperationsschemas*** einer Gruppe von Klassen
  - ***Homogenität*** unter verschiedenen Anwendungssystemen einer Problemfamilie (z.B. ähnliche, ergonomische Bedienung)
  - Ein spezifisches Anwendungssystem der Problemfamilie entsteht aus dem Framework durch ***Instanziierung und Ergänzung***.

Vergleich Klassenbibliothek-Framework

Klassenbibliothek



Framework



Anwendungsspezifische Teile

Vorgefertigte Teile

"Don't call us, we call you"

Ablaufsteuerung

nicht vordefiniert

im wesentlichen vordefiniert

Die Grenze ist fließend, siehe z.B. Java AWT !

Anpassung von Frameworks

- ***Objektorientierte Techniken zur Anpassung:***
  a) Neue **Unterklassen** von (abstrakten oder konkreten) Framework-Klassen
     - **Entwurfmuster Template Method**
  b) Instanziierung und **Konfiguration** von Objekten
  c) Methodenaufruf und **Festlegung von Parameterwerten**
- ***Anpassungsstelle eines Frameworks (Hot Spot):***
  – ***Vorbereitete*** Anpassungsmöglichkeit
  – Für den Entwickler ***dokumentiert***
  – Umfaßt oft mehrere Anwendungen obiger Techniken
     - z.B. mehrere Einschubmethoden (***Hooks***) für eine Anpassungsstelle (*Hot Spot*)
- Aber: wenn verteilte Änderungen durchzuführen sind, ist die Wiederverwendung sehr schwierig
- Problem: API muss stabil bleiben → schwierige Weiterentwicklung der Klassenbibliothek

Arten von Frameworks

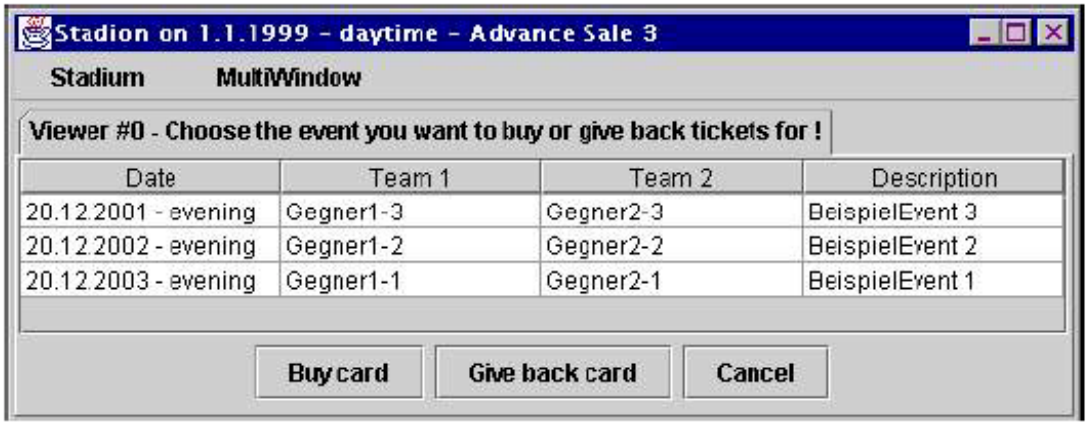
- ***Arten der Wiederverwendung:***
  - mit Kenntnis der Struktur: ***White box reuse***
  - ohne Kenntnis der Struktur: ***Black box reuse***

- ***Arten von Frameworks:***
  - **Application / Support Framework** = Architektur für Infrastruktur einer Anwendung
- **Domain Framework** = Expertenwissen eines fachlichen Gebiets
- Betrachtete Beispiele: ***Tasking*** / ***Swing*** / ***Grafik***
- Frameworks in verwandten Domains:
  - ***SalesPoint*** / San Francisco / JWAM

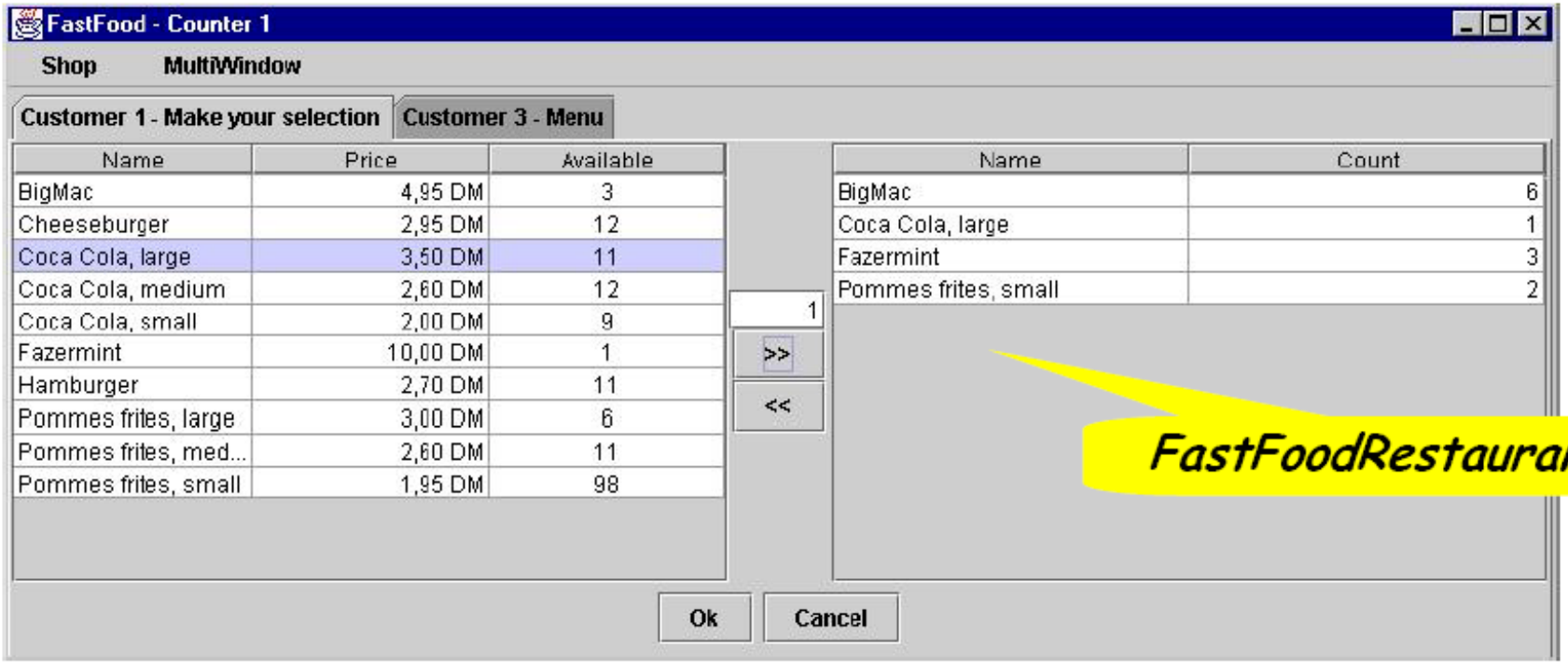
Beispiel: Das Framework SalesPoint

- *SalesPoint*
  - ist ein ***Framework zur Erstellung von Verkaufsanwendungen***;
  - stellt Mechanismen zur Verwaltung von ***Katalogen***, ***Beständen***, Währungen, Benutzern und Datenkörben bereit;
  - unterstützt die Entwicklung von ***Verkaufsprozessen***:
    - ***Abläufe*** in einem Geschäft („Shop“), in dem Kunden an einer Reihe von Verkaufsstellen oder Schaltern („SalesPoint“) bedient werden;
    - SalesPoint-Kunden führen ***„Datenkörbe“*** („data baskets“) mit sich, welche die vom Kunden ausgewählten Waren enthalten (ein Datenkorb ist Gegenstand einer ***Einkaufstransaktion***);
    - das ***Speichern und Laden*** von „Spielständen“;
    - das ***Mitverfolgen*** *(logging)* von Ereignissen in einem Geschäft;
  - stellt ***domainspezifische GUI-Komponenten*** bereit. (z.B. Auswahl aus Katalog)

SalesPoint 2.0: Beispiele



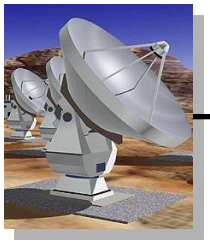
Fussballstadion

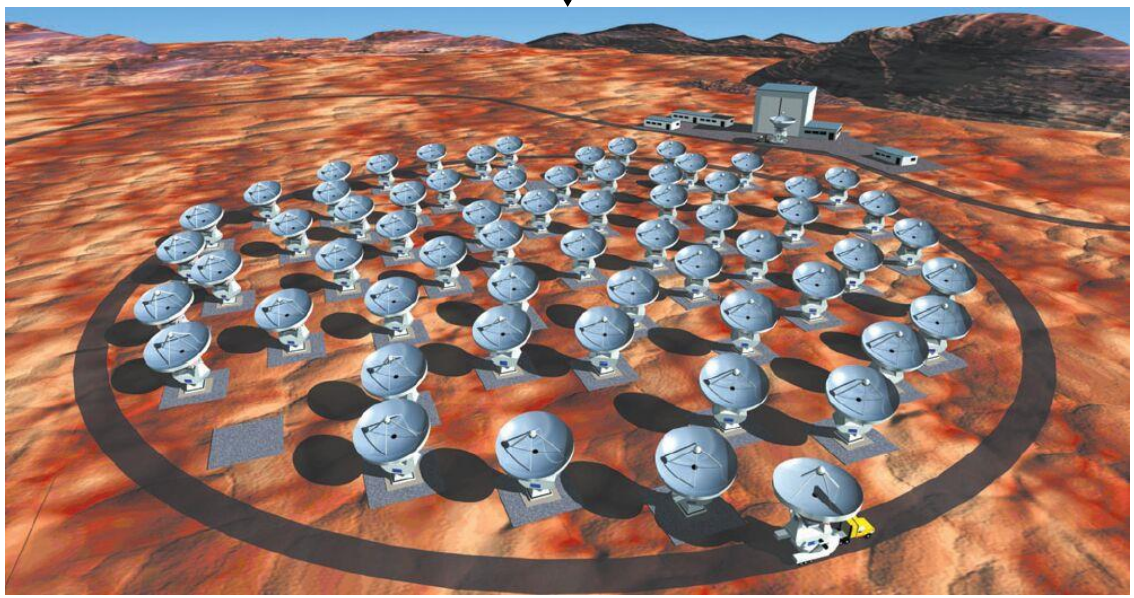


FastFoodRestaurant

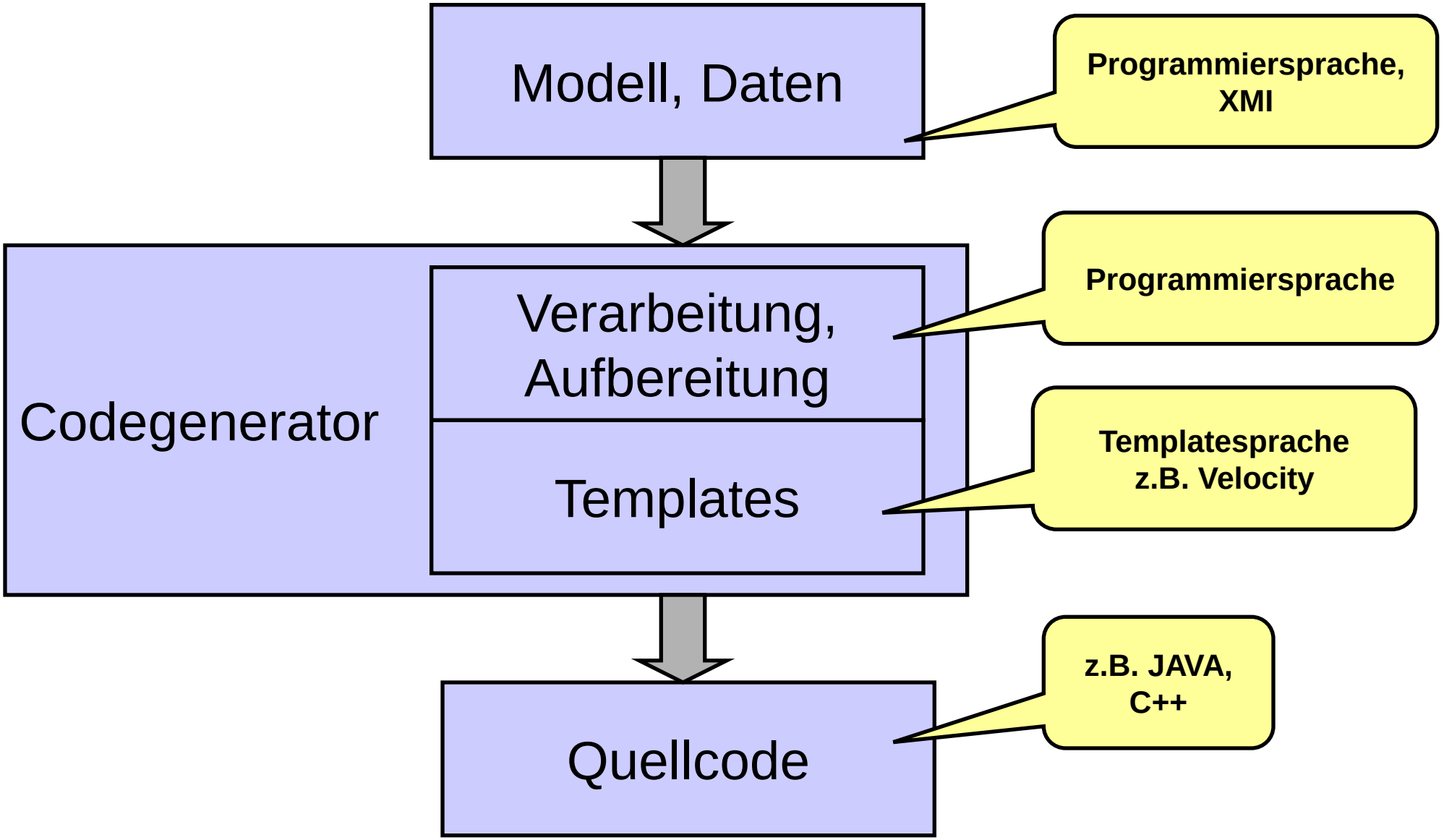
Fragen ?

9.10 Codegeneratoren

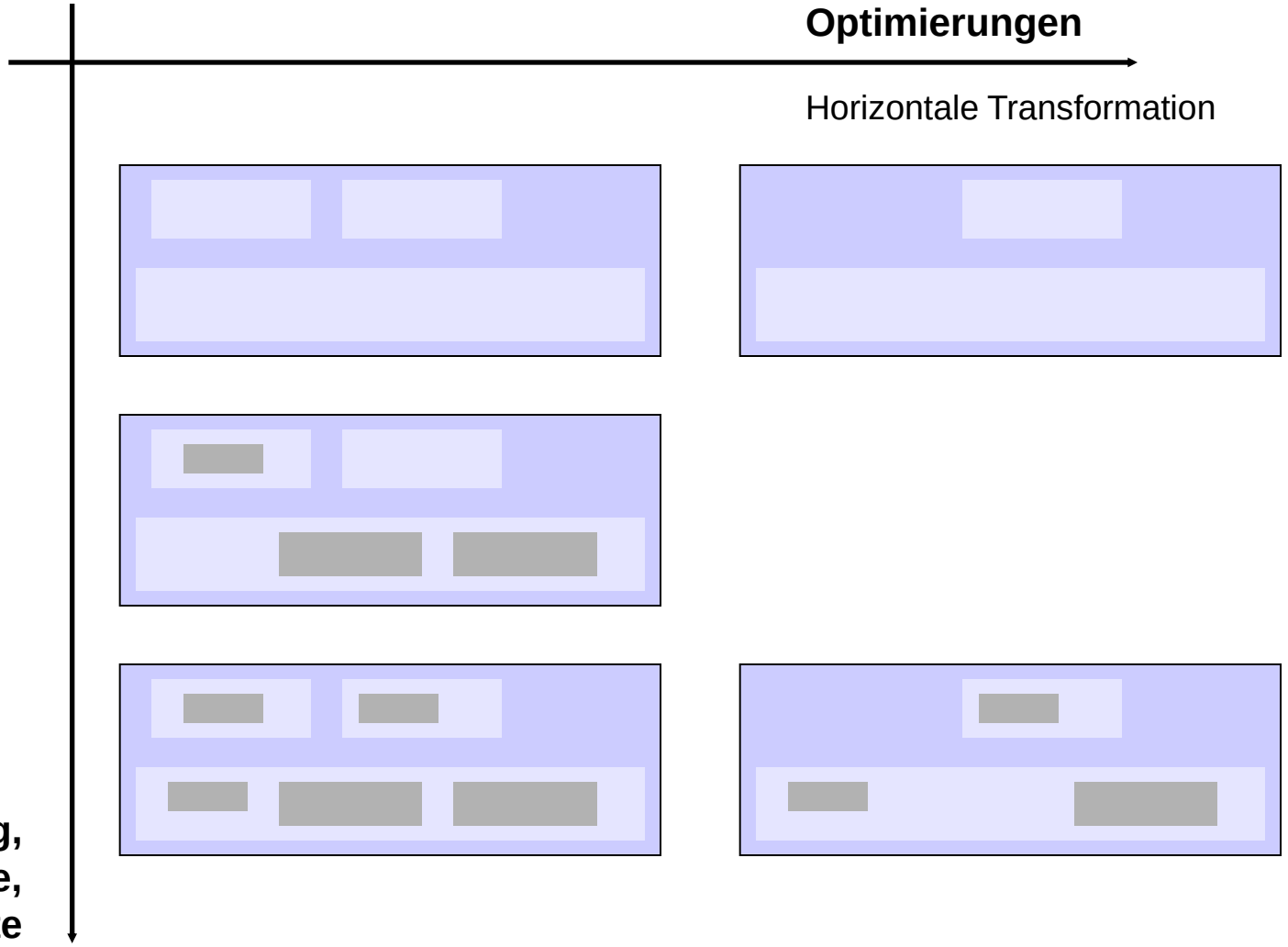




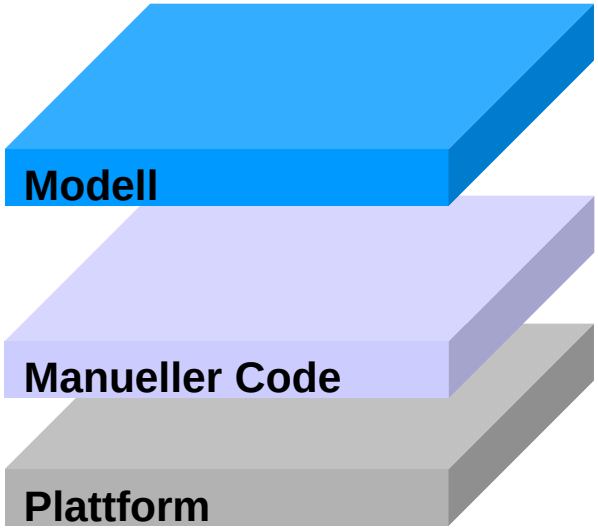
Architektur

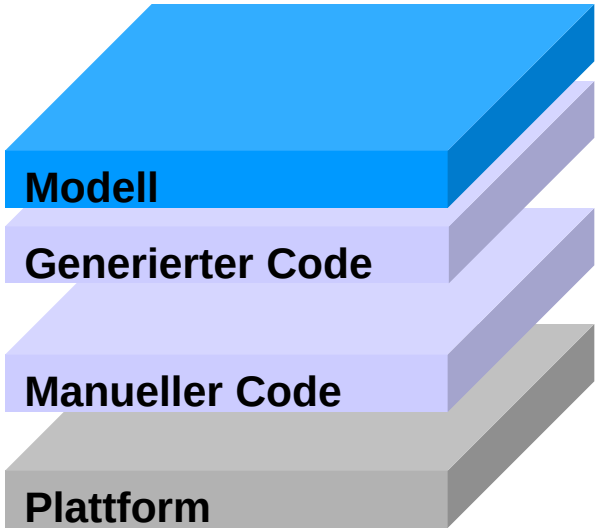


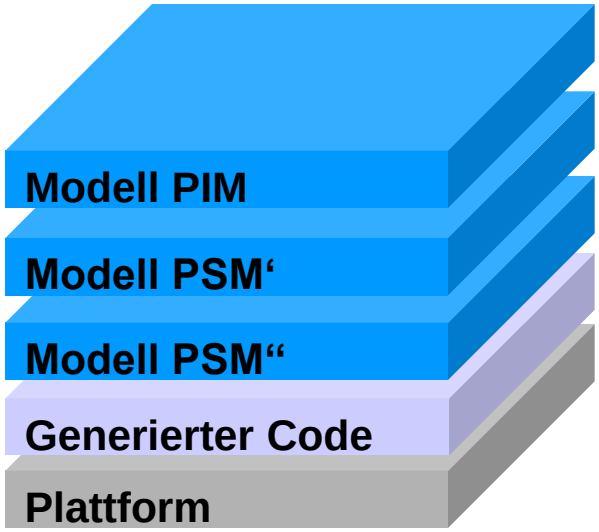
Transformationsraum

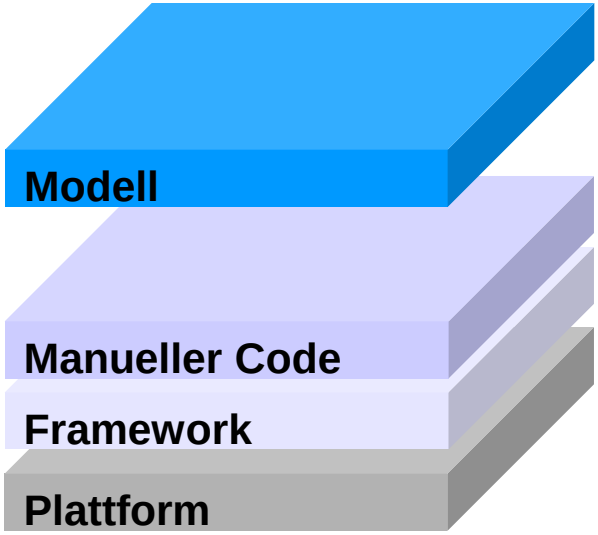


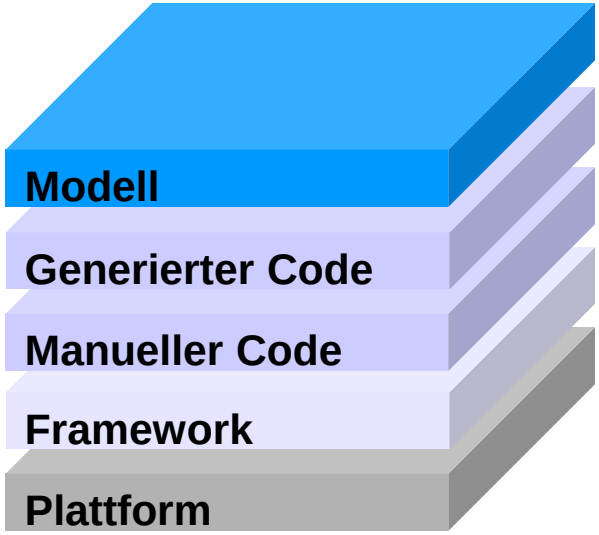
Vom Modell zum plattformspezifischen Code

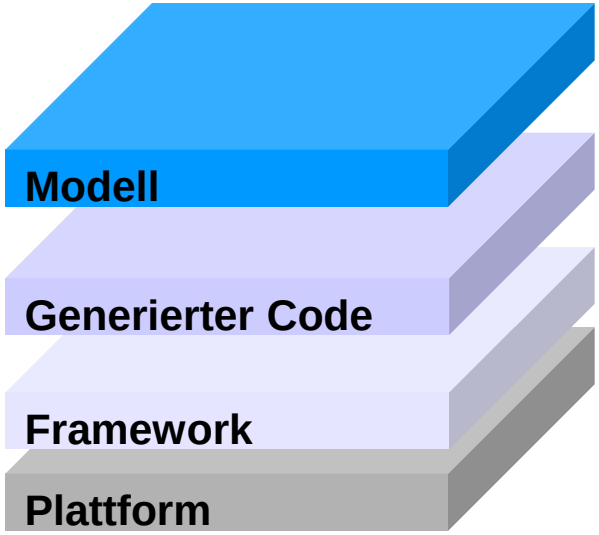






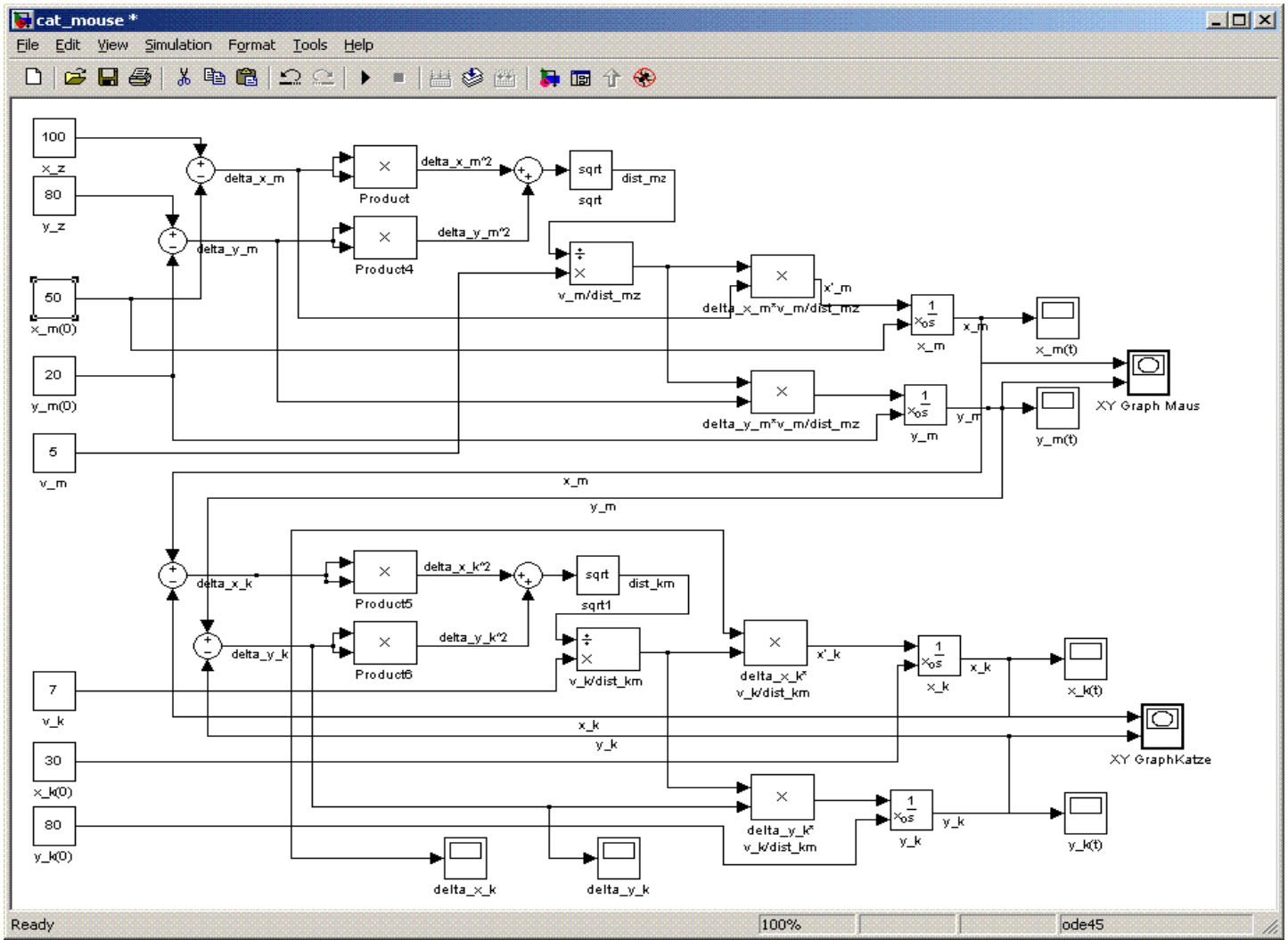


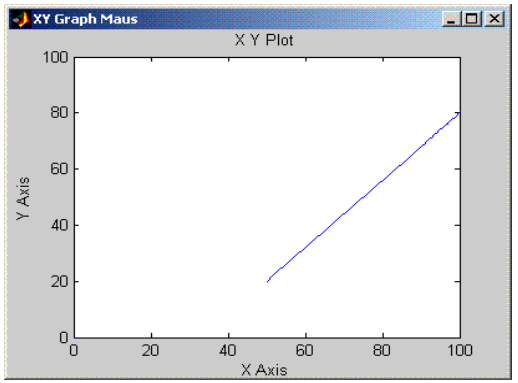


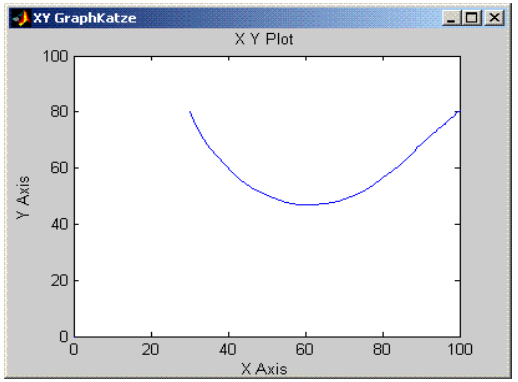


Beispiel: SimuLink

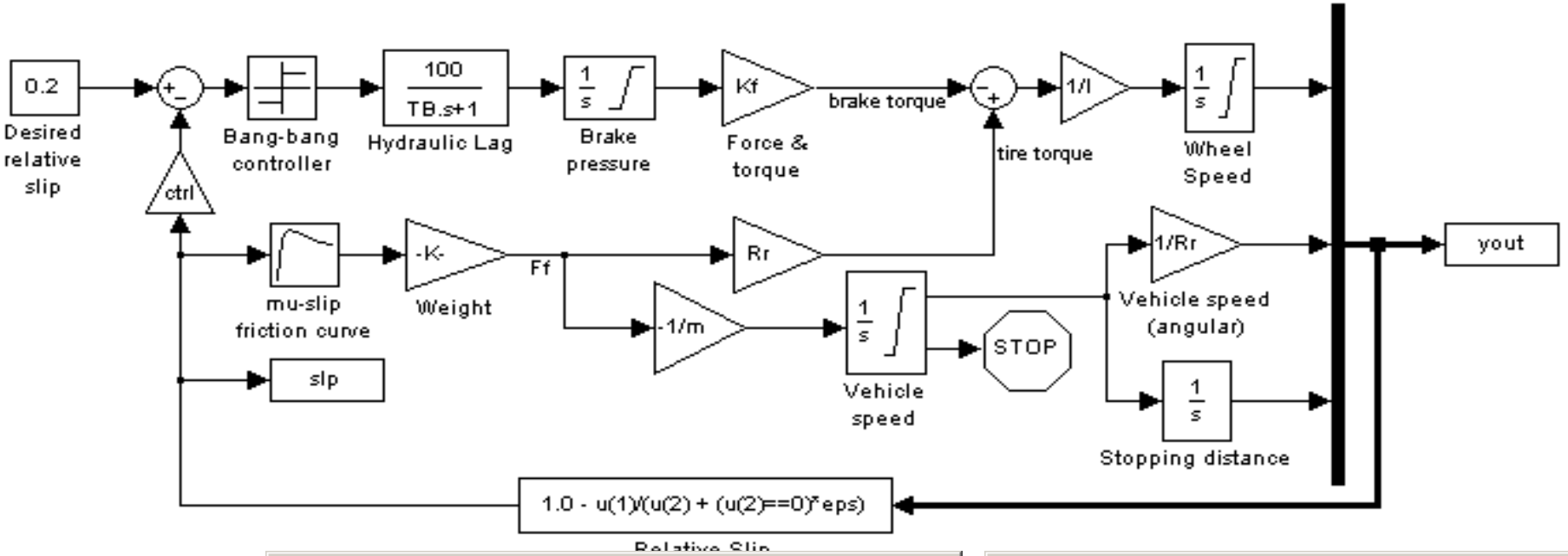
- Modellierung durch Datenflussdiagramm
  - jede „Leitung“ entspricht einer Variablen



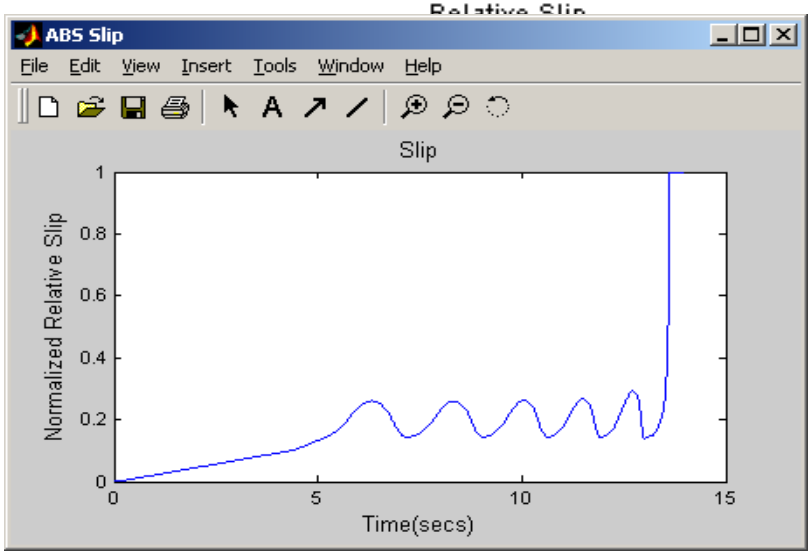


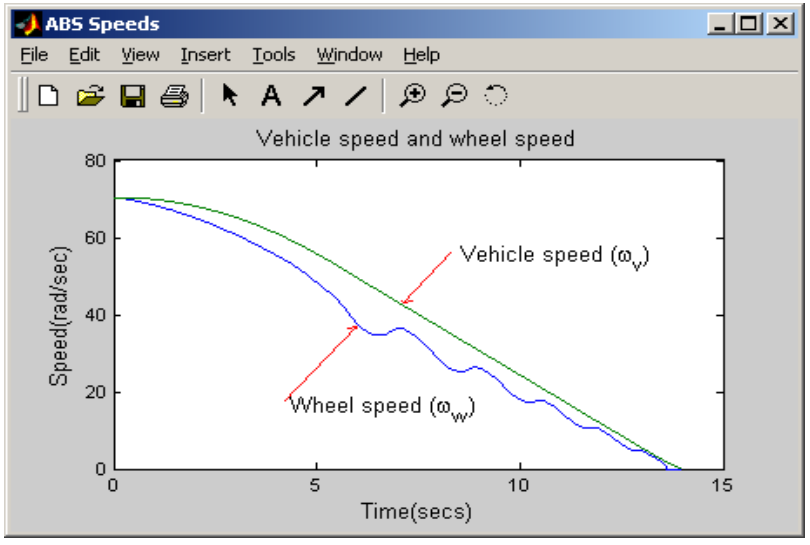


ABS Braking Model



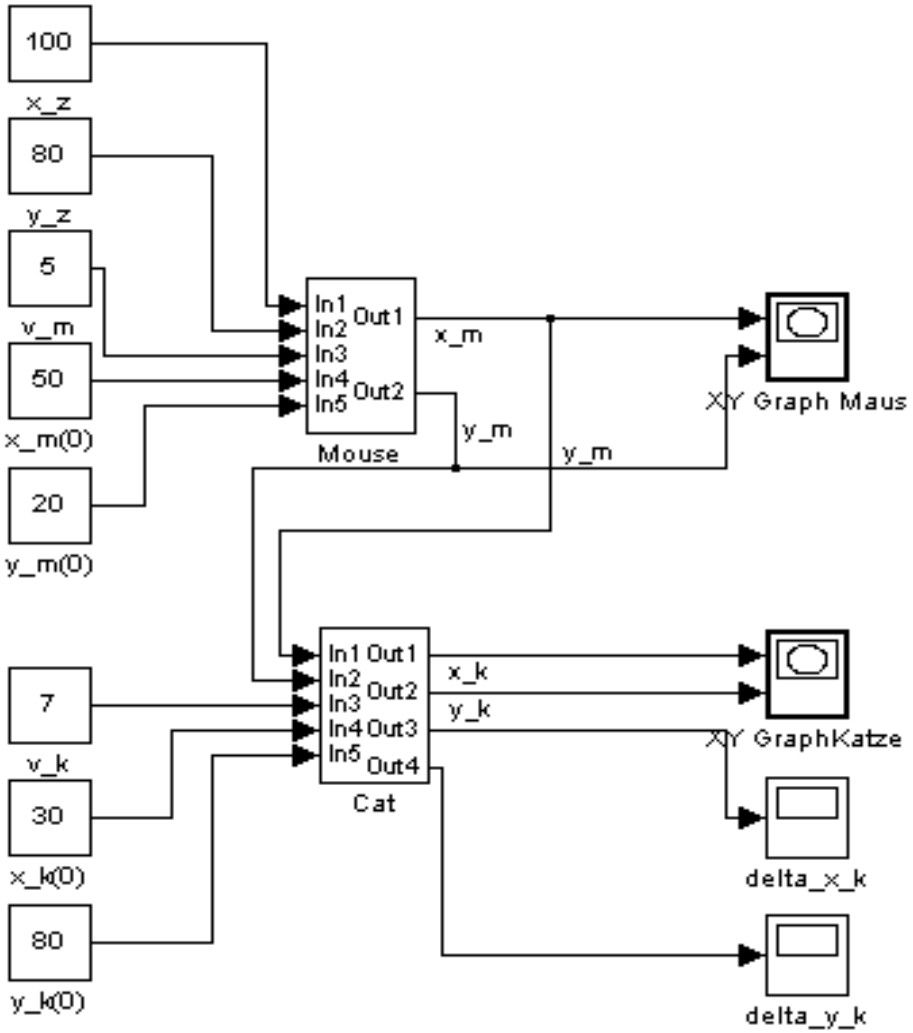
Double click to run model and plot the results



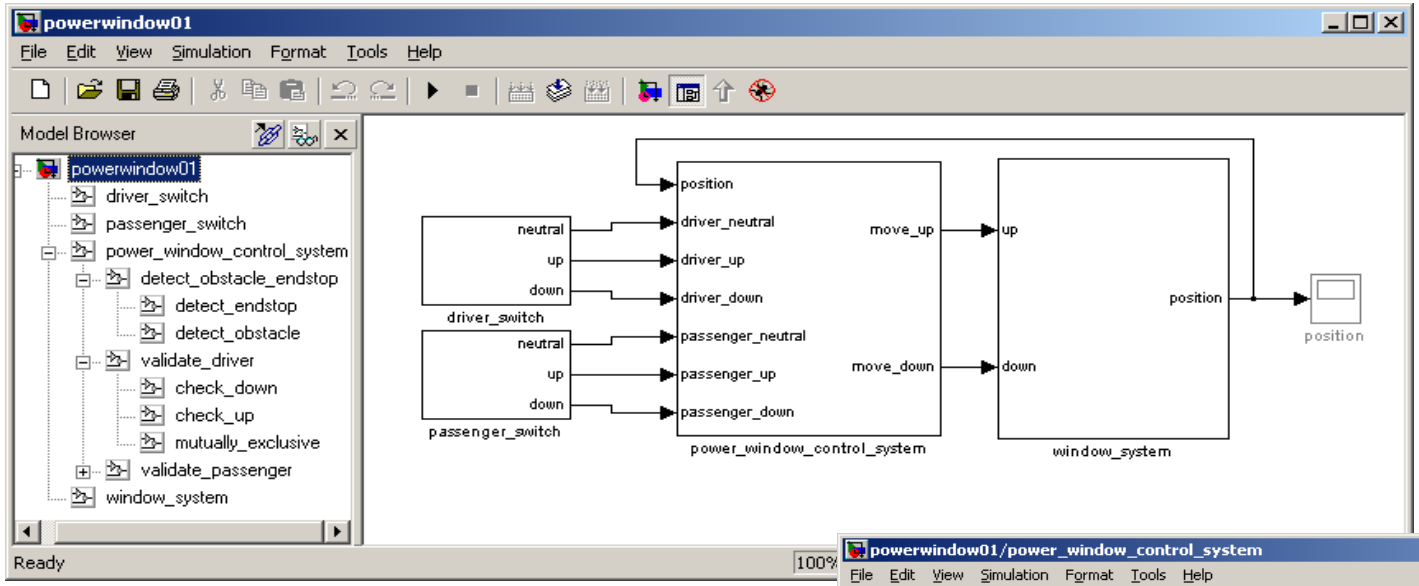


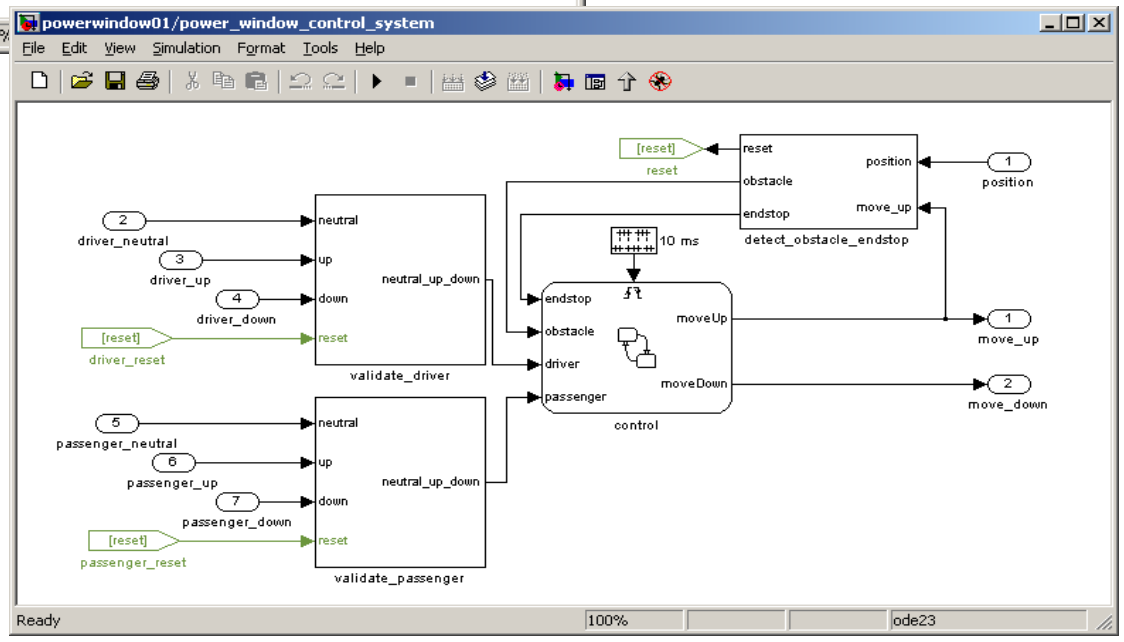
Abstraktion

- Hauptstärke von SimuLink besteht in der Möglichkeit, Blöcke zusammenzufassen
  - Abstraktion von Verhalten
  - baumartige Navigation
  - Parametrisierung
  - Modulbibliotheken
  - externe Erweiterungen
  - Codeanbindung
- Modelltransformation und –entwicklung!

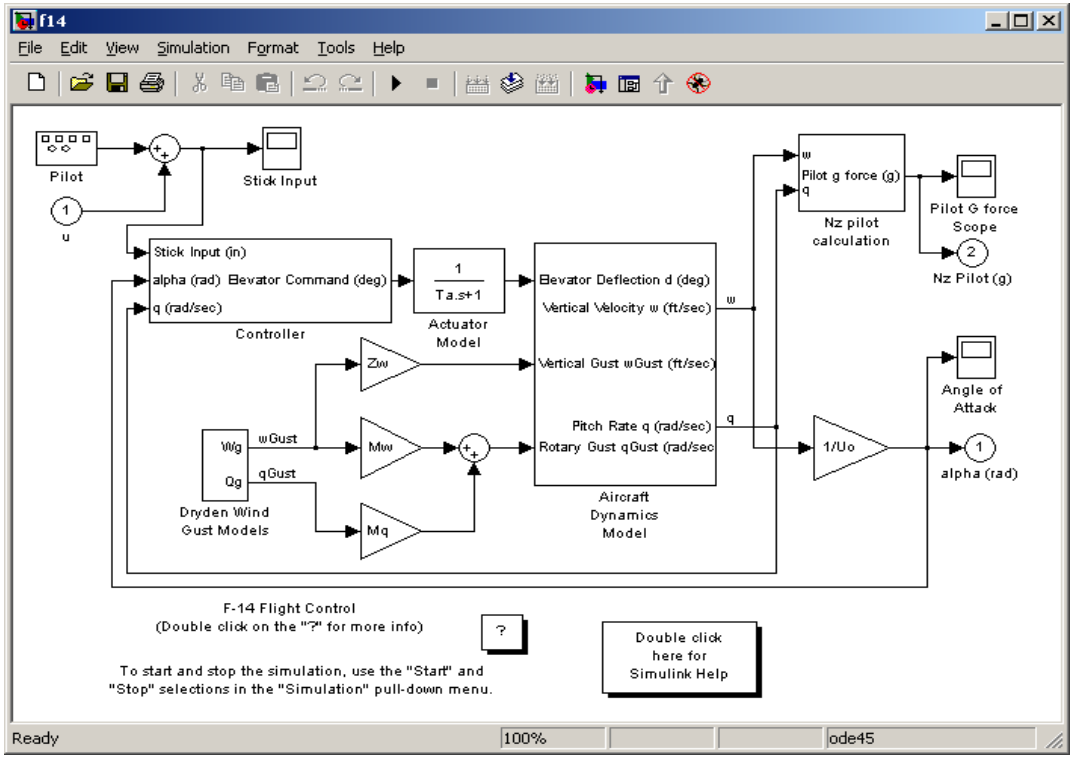


Beispiel: Fensterheber



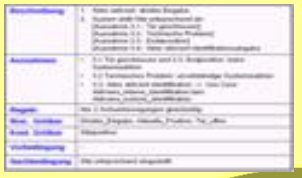


- Frühzeitige Erstellung eines ausführbaren Modells (Systemmodell) auf Grundlage der Anforderungen
- Test und Erprobung auf Modellebene
- Verfeinerung zum Implementierungs-modell
- Automatische Code-generierung für Host und Target
- Automatische Testgenerierung



Aktivitäten und Artefakte

Lastenheft



Formalisierung

formale Spezifikation

$\Box(p^{\circ} - p^{\bullet} \leq 5)$

$\Box(k - \varepsilon \leq |poll^{\bullet} - poll^{\circ}| \leq k + \varepsilon)$

Virtueller Prototyp

Prototyping

Modellierung

Systemmodell

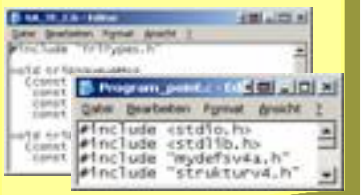
Verifikation

Testgenerierung

Testgenerierung

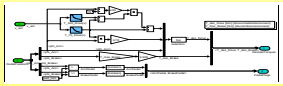
Testfälle

Zusätzlicher Code



Verfeinerung

Implementierungs- modell



Codegenerierung

Testausführung

Target-Code

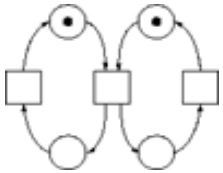
Integration

Portierung

Zielsystem

Modellierungsformalismen

- Herkömmliche Formalismen: Zustandsmaschinen, Diagramme, Petrinetze, StateCharts, Message Sequence Charts, ...
  - gut etabliert, Toolunterstützung vorhanden
  - Wildwuchs, Varianten, mangelnde Konstanz
  - Prognose: werden durch UML2.0 abgelöst werden



- UML 2.0
  - vereinigt verschiedene Arten von Diagrammen
  - extrem mächtig, sehr umfangreich zu lernen
  - fehlende Semantik, verschiedene Ausbaustufen

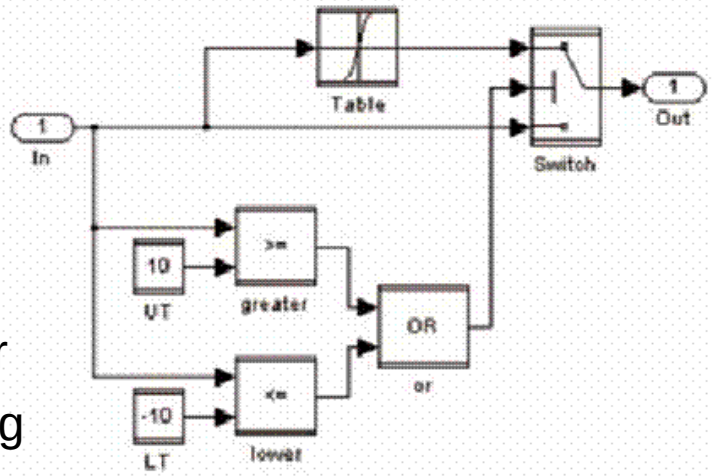
- Matlab / Simulink / Stateflow
  - teilweise gut eingeführt und bekannt
  - limitierte Ausdrucksfähigkeit
  - nur eine Quelle

- Logische und algebraische Modelle, z.B. TLA, ASM/ASML, CSP
  - nahe an natürlichsprachlichen Formulierungen
  - Forschungsbedarf

„Bist du sicher, dass du kein Modell brauchst?“

Codegenerierung

- Ziel: automatische Übersetzung von Modellen in ausführbaren (C-) Code
- Zwei kommerzielle Produkte verfügbar
  - Real Time Workshop (The MathWorks)
  - TargetLink (dSPACE GmbH)
- Codegenerator ist „Compiler für Modelle“
  - Wiederverwendung
  - schnelle Prototyp- und Produkterstellung
  - erhöhte Zuverlässigkeit gegen Programmierfehler
  - automatische Optimierung des generierten Codes
- Wie kann man sicherstellen, dass der generierte Code das Erwartete leistet?



* Original code with type conversion operators
oved for better readability

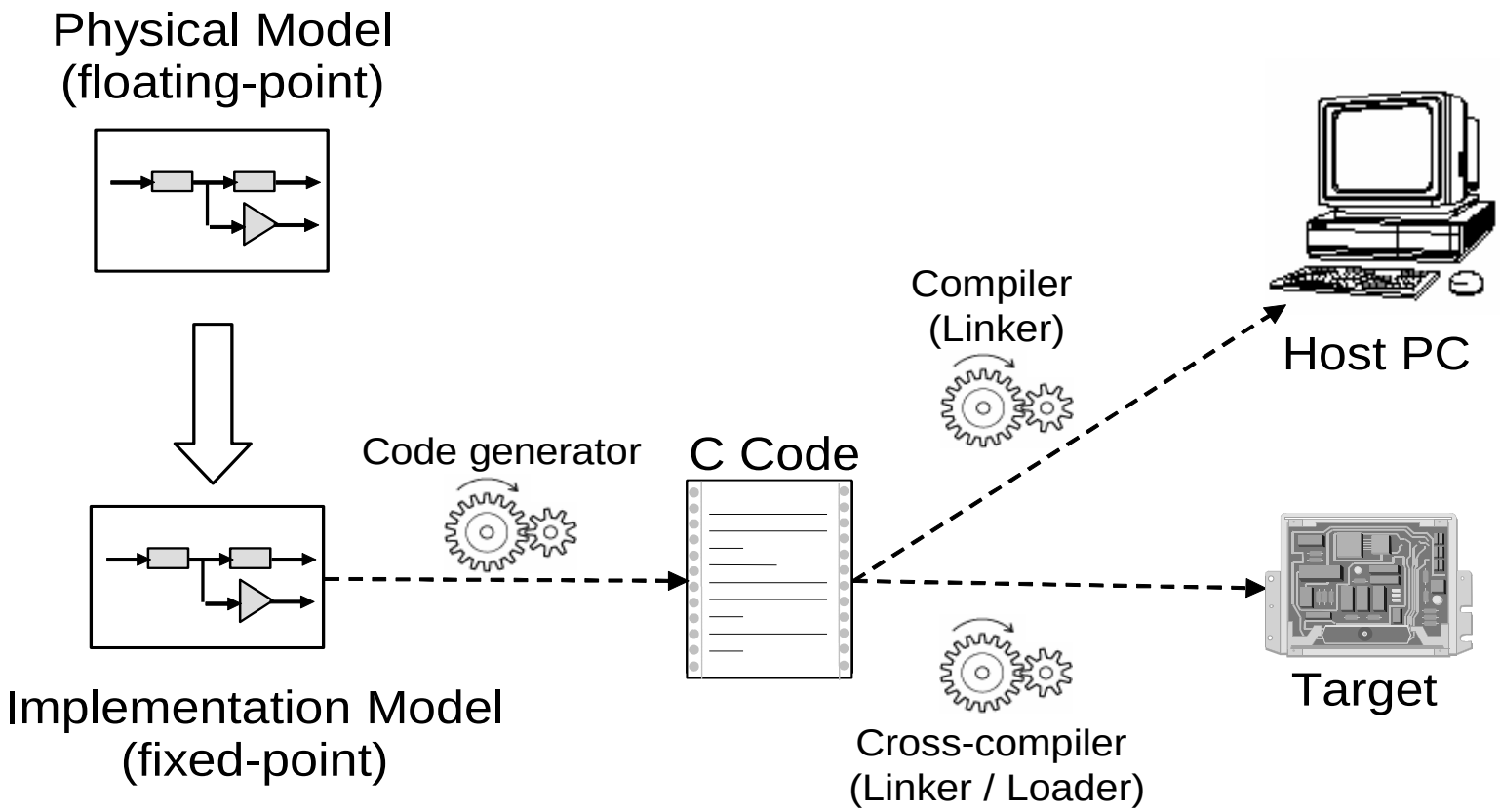
Non-optimized Code:

```
bool1 = (In >= 10);
bool2 = (In <= -10);
bool3 = bool1 || bool2;

tmp1 = table_lookup(Table, In);
tmp2 = in;

if (bool3)
    out = tmp1;
else
    out = tmp2;
```

TargetLink Code:

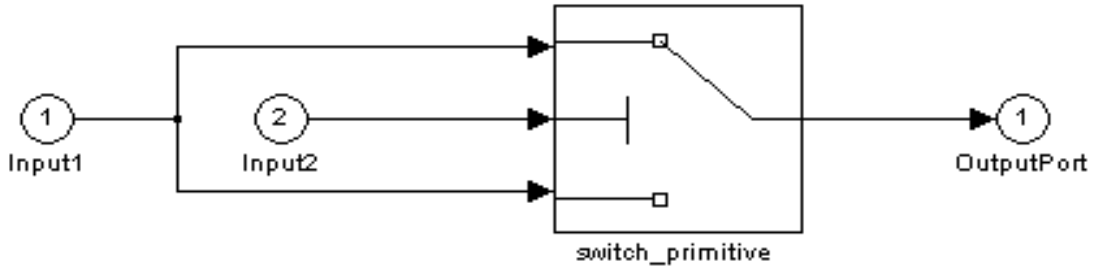
```
if ((in >= UT) || (in <= LT))
    out = table_lookup(Table, in);
else
    out = in;
```

Prinzip

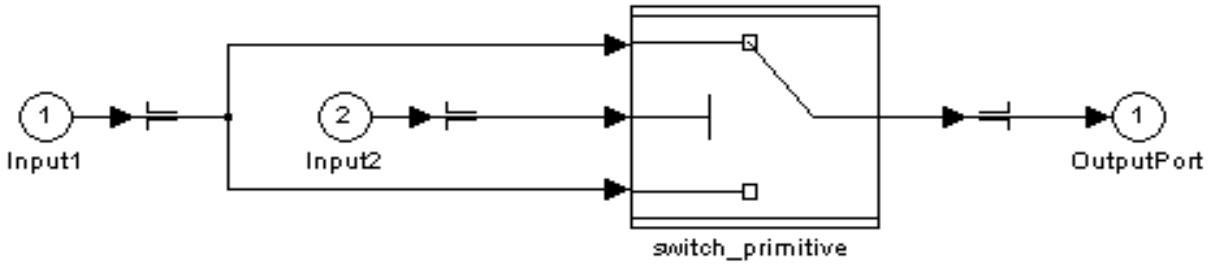

Physical Model
(floating-point)
Implementation Model
(fixed-point)
Code generator
C Code
Compiler
(Linker)
Host PC
Target
Cross-compiler
(Linker / Loader)

Beispiel

- Schaltsystem mit Schwellwert 0.5
  - physikalisches Modell (SimuLink)



  - Implementierungsmodell (TargetLink)



  - Äquivalenz?

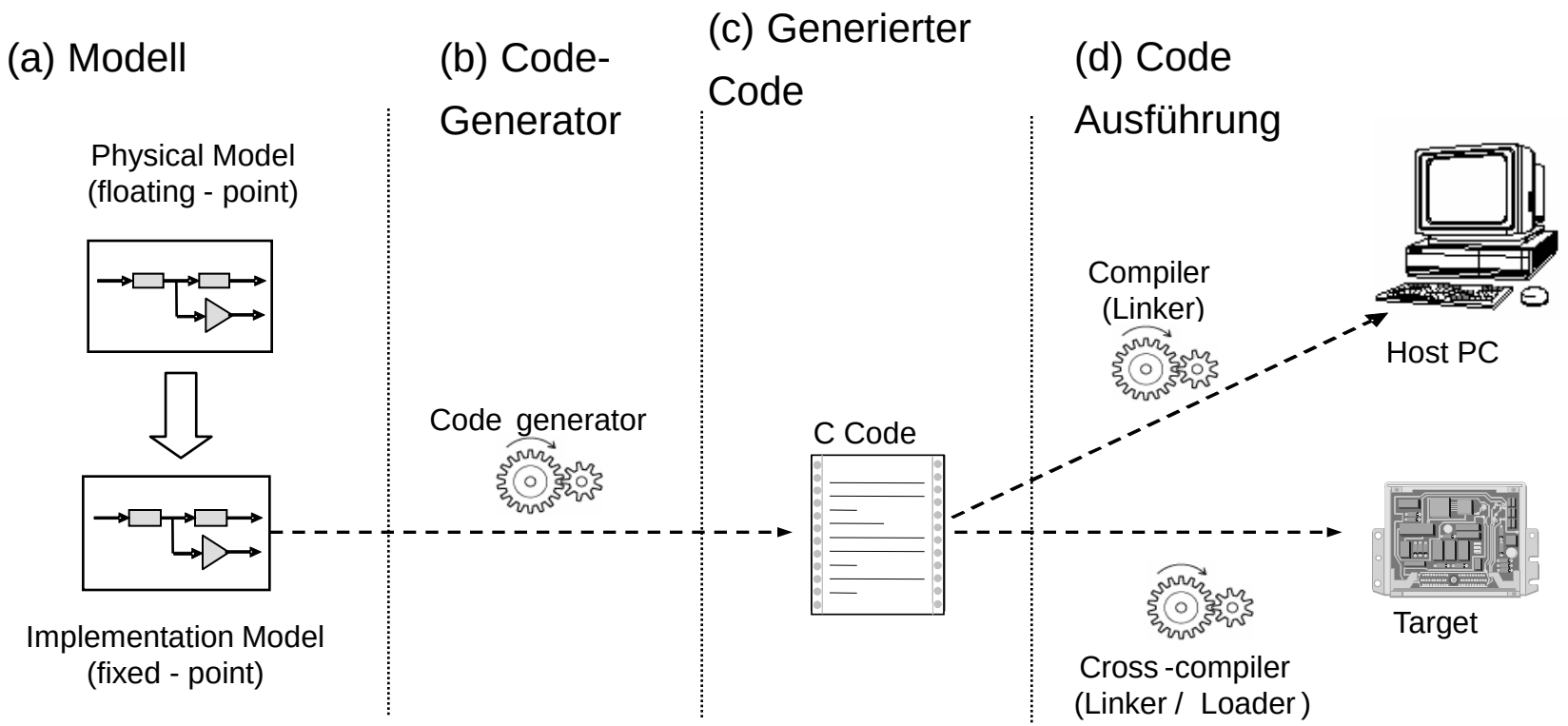
generierter Code

- Zweierpotenz-Skalierung; $128 * 2^{-8} = 0.5$

```
Void switch_system(Void) {
    /* Switchswitch_system/switch_primitive */
    if (Sa1_Input2_ >= 128 /* 0.5 */) {
        /* # combined # Outport: switch_system/OutputPort */
        Sa1_OutputPort_ = Sa1_Input1_;
    }
    else {
        /* # combined # Outport: switch_system/OutputPort */
        Sa1_OutputPort_ = Sa1_Input1_;
    }
}
```

Codeabsicherung

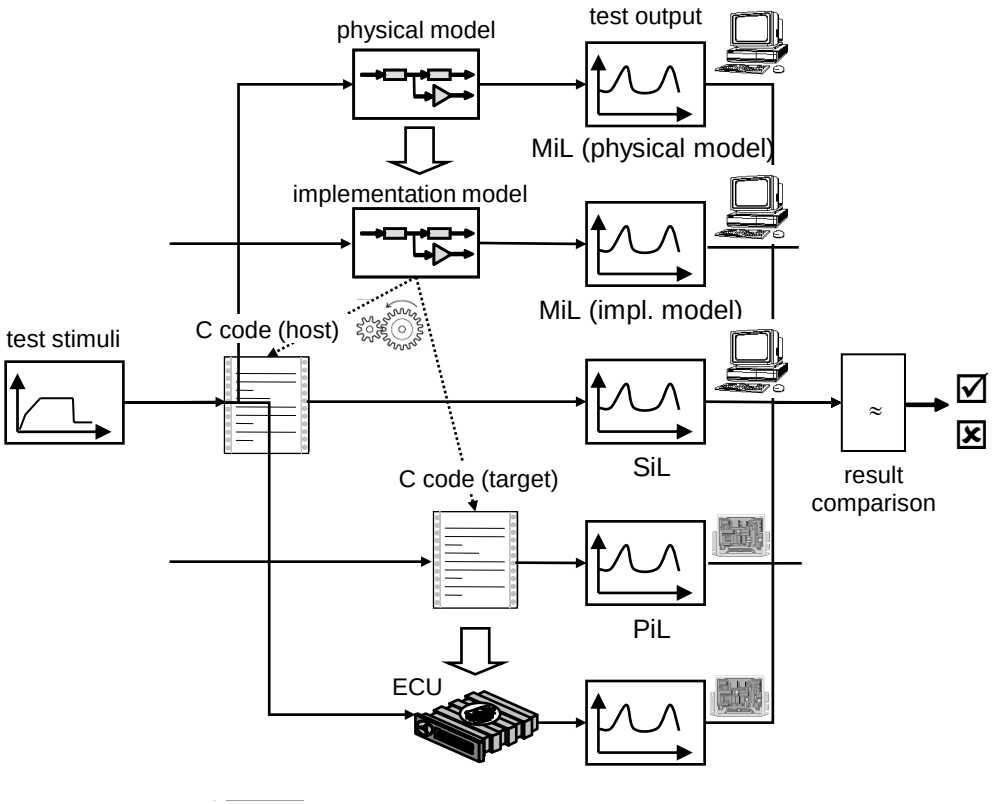
- Qualitätssicherung auf jeder Ebene notwendig!



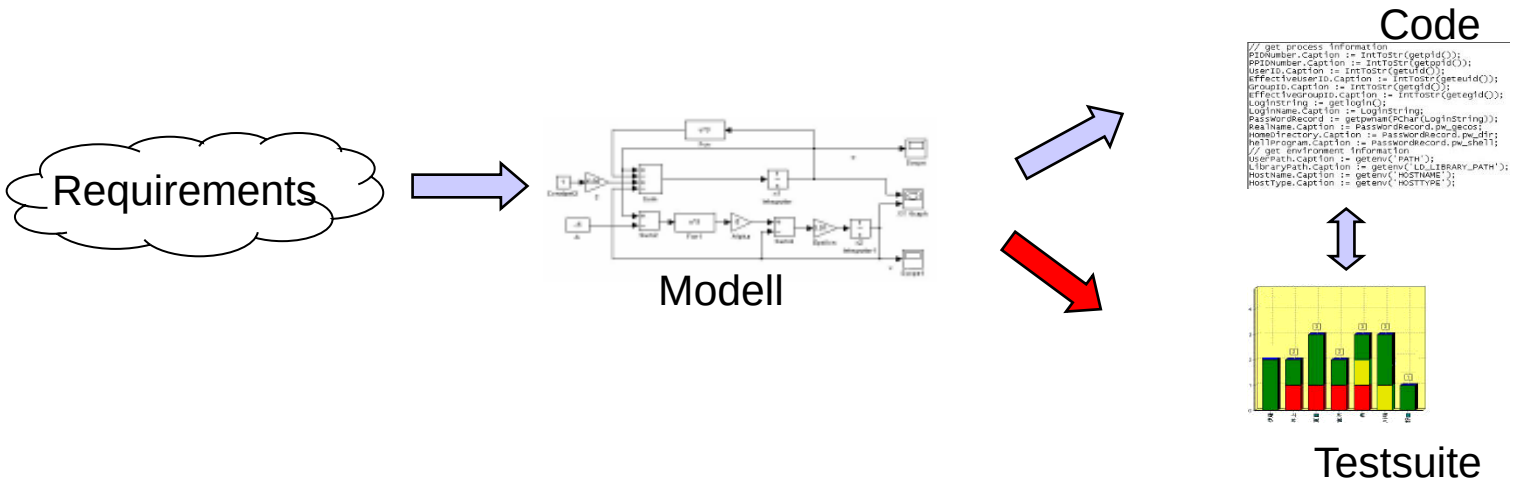
QS auf Modellebene

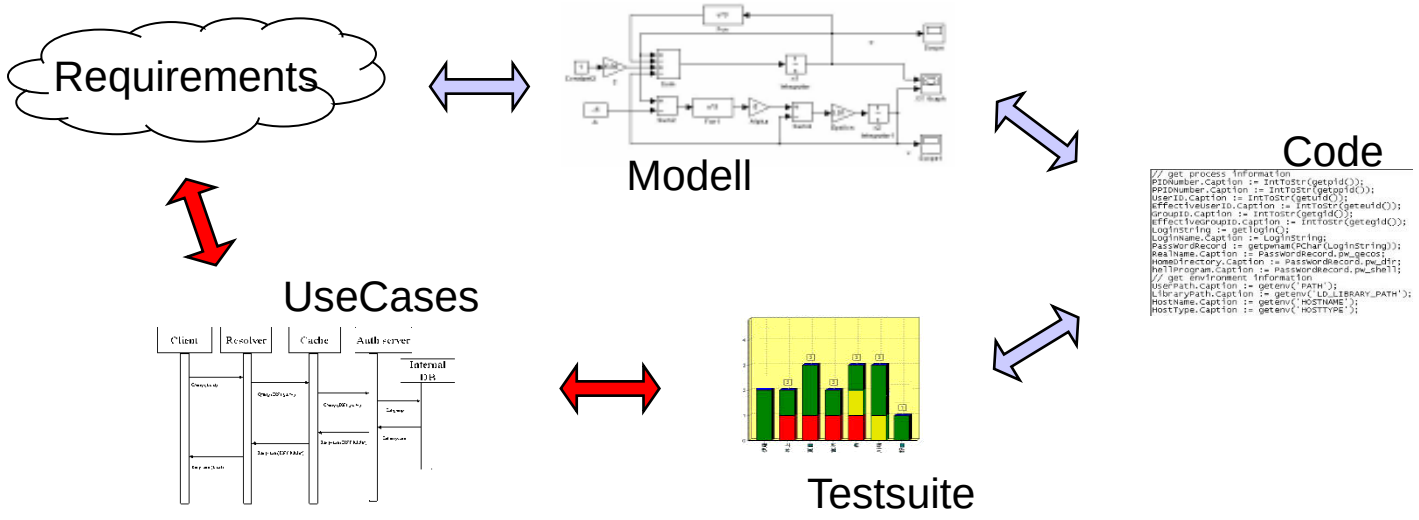
- Modellierungsrichtlinien
  - Qualität des Implementierungsmodells ausschlaggebend für Qualität des generierten Codes
  - Richtlinien und Muster existieren (z.B. MathWorks Automotive Advisory Board - MAAB guidelines)
  - Toolunterstützung zur Umsetzung der Richtlinien
    - demnächst www.eGuidelines.de
  - Vorteile
    - Lesbarkeit
    - Wartbarkeit
    - Wiedervervendbarkeit
    - Testbarkeit

- Simulation /Ausführung des Modells und generierten Codes in verschiedenen Entwicklungsphasen
  - MiL
    (Model in the Loop)
  - SiL
    (Software in the Loop)
  - PiL
    (Processor in the Loop)
  - HiL
    (Hardware in the Loop)

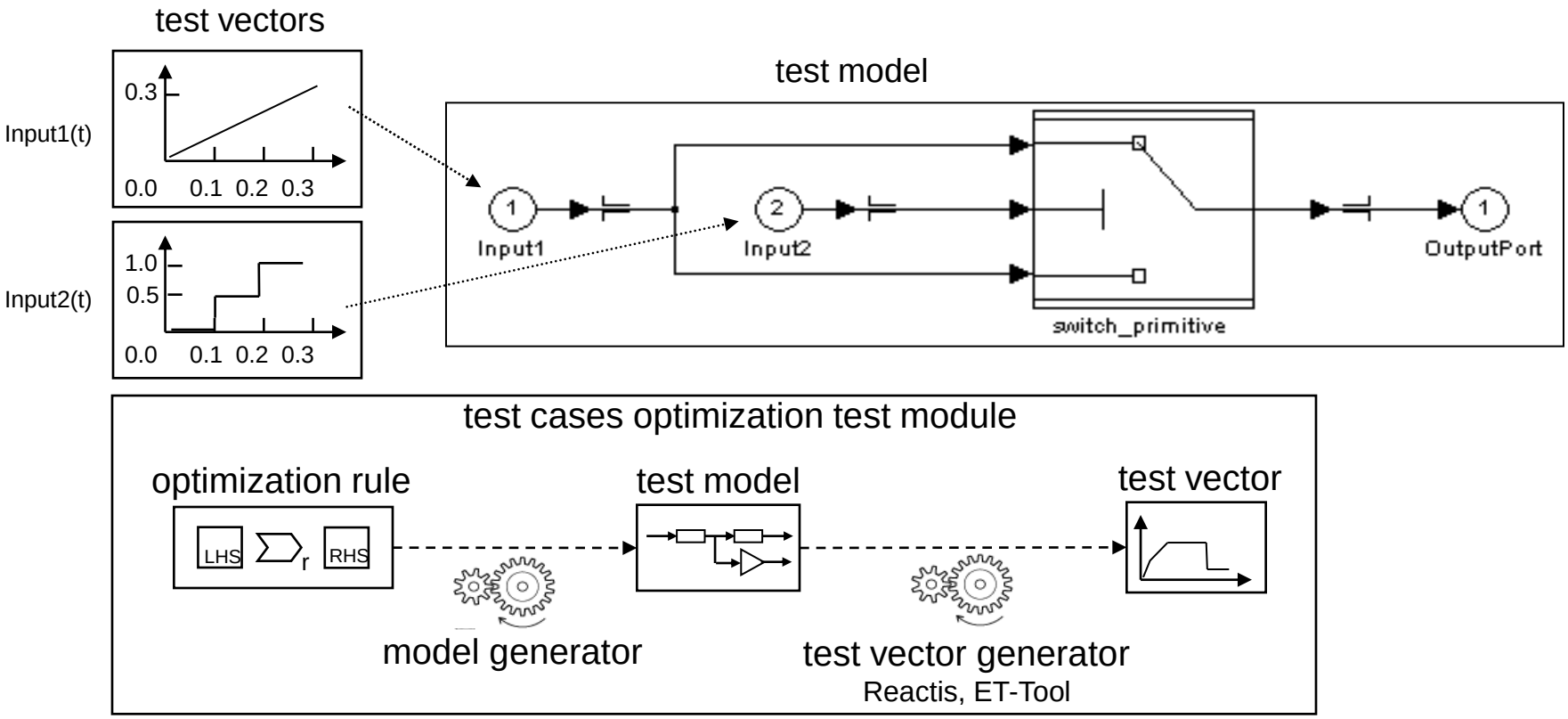


Szenarien für Testautomatisierung





Im Beispiel


test vectors
Input1(t)
0.3
0.0 0.1 0.2 0.3
Input2(t)
1.0
0.5
0.0 0.1 0.2 0.3
test model
1
Input1
2
Input2
switch_primitive
1
OutputPort
test cases optimization test module
optimization rule
LHS
r
RHS
test model
test vector
model generator
test vector generator
Reactis, ET-Tool

Fragen ?