Software Engineering

Implementierung

Prof. Dr. Peter Jüttner

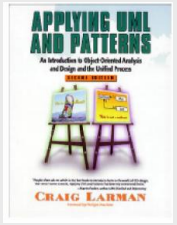
Hochschule Deggendorf

Inhalt

Literatur zur Vertiefung

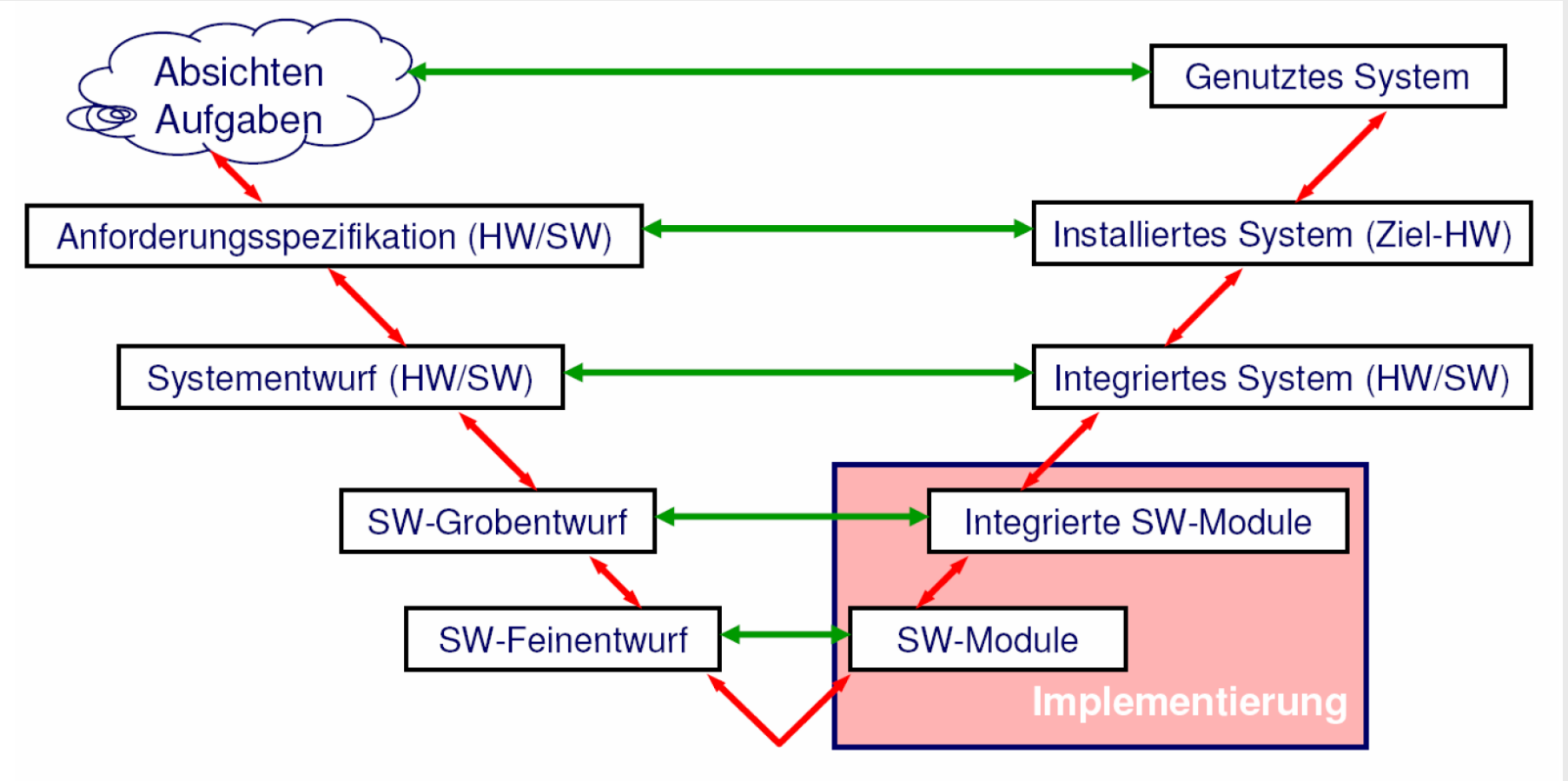
- Helmut Balzert
  Lehrbuch der Software-Technik, Teil 1
  Spektrum Akademischer Verlag, 2001

- Helmut Balzert
  Lehrbuch der Software-Technik, Teil 2
  Spektrum Akademischer Verlag, 1998



- Craig Larman
  Applying UML and Patterns
  Prentice Hall PTR, 2002

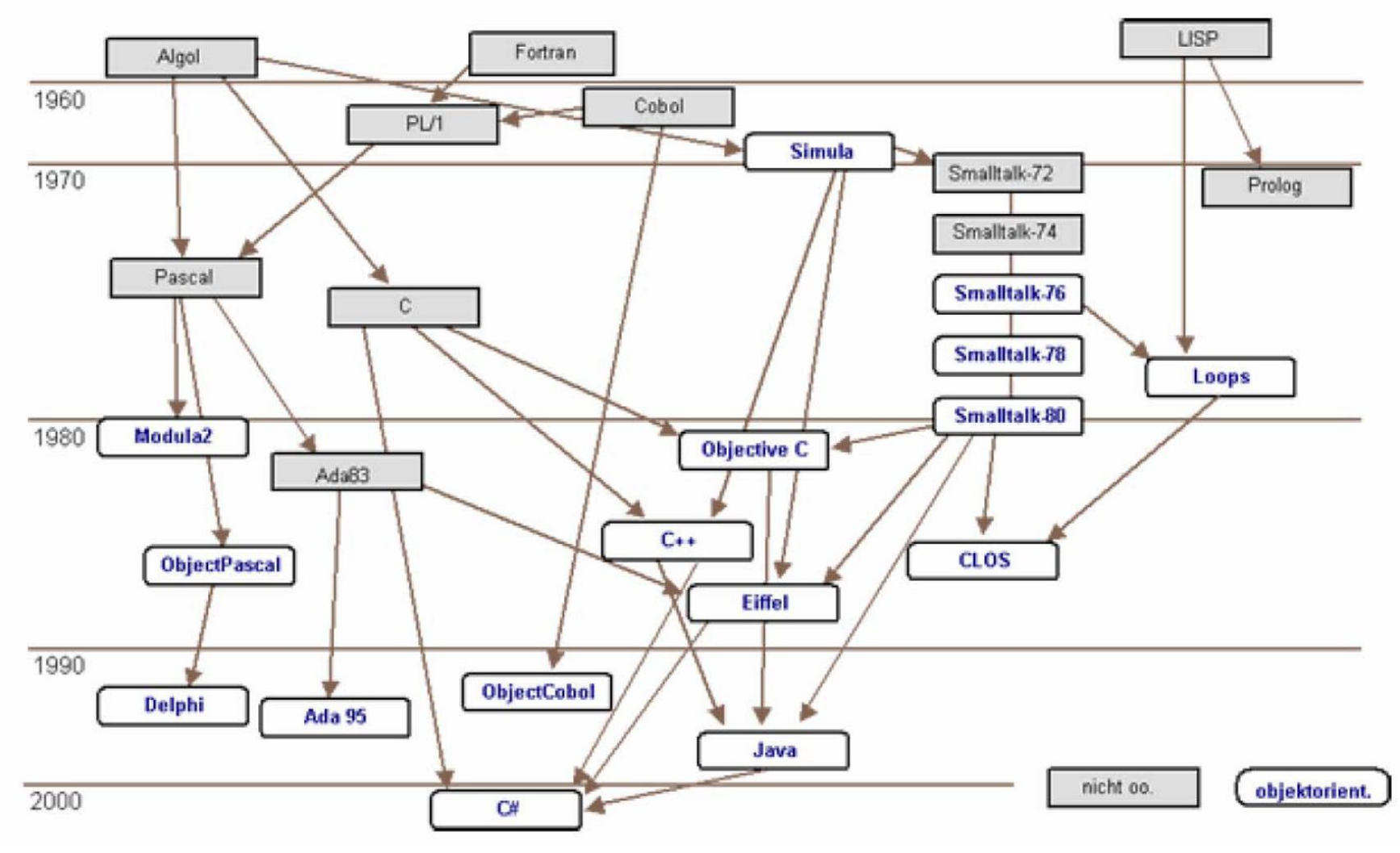
Bereich des V-Modells



Implementierungsphase

- Wahl der Programmiersprache
  - die Programmiersprache ist oft im Auftrag spezifiziert
  - zumindest eine Klasse von Programmiersprachen (typischerweise Paradigma) sollte vorweg festgelegt werden
- Kosten-Nutzen-Analyse
  - bestimme Kosten und Nutzen der in Frage kommenden Programmiersprachenkandidaten, also insbesondere
  - die Notwendigkeit der Einstellung von neuem Personal
  - die Weiterbildung des vorhandenen Personals

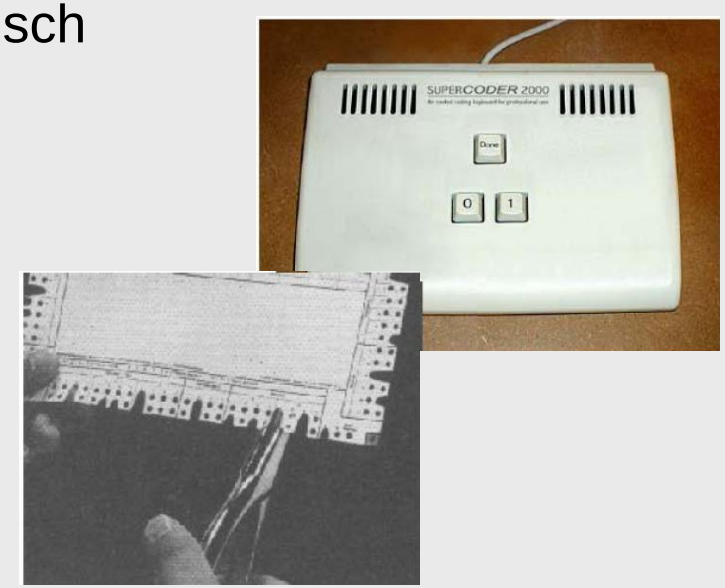
Historische Entwicklung



Programmiersprachen-Generationen

- 1. Generation:
  - „Computer“ kennen nur zwei Zustände:
    - Strom an („1“)
    - Strom aus („0“)
  - Programme („Kochrezepte“) waren Abfolgen binärer Befehle, meist über Schalter mechanisch einzustellen
  - z.B.:

```
00110101 10101101 00110101
00110011 10101101 00110101
10101101 01100110 00110101
00110101 10101101 00110101
01100110 10101101 00110101
10101101 01100110 00110101
```



Programmiersprachen-Generationen

- 2. Generation:
  - Benennung solcher „maschinennaher“ Befehle durch "Namen" (symbolische Notation — Assembler)
  - benötigt Übersetzung in „Maschinensprache“ (0/1): 1-zu-1
  - z.B.

```
mov  dx,03c4h
mov  ax,0604h
out  dx,ax
neq  ax,0100h
out  dx,ax
lda  dx,03c2h
mov  al,0e3h
neq  dx,al
mov  dx,03c4h
add  ax,6500
ret
```

Programmiersprachen-Generationen

- 3. Generation:
  - weitere Abstraktion: „höhere“ Programmiersprachen
  - Befehle intuitiver, „Kochrezept“ näher an der natürlichen Sprache
  - benötigt kompliziertere Übersetzer
    - 1-zu-5 oder 1-zu-10 Maschinenbefehle
  - z.B. COBOL, FORTRAN, Pascal,..., C++, Java

```
for (int i=1; i < surveyor.length; i++) {
  if (surveyor[i].rating.equals(excellent)) {
      surveyor[i].salary += 6500;
  }
}
```

Programmiersprachen-Generationen

- 4. Generation:
  - 1-zu-30 oder 1-zu-50 Maschinenbefehle
  - z.B. Focus, Natural

```
for every surveyor
  if rating is excellent
     add 6500 to salary
```

Unterschiede C vs. C++/Java

- C ist funktionale Programmiersprache
  - gut für Embedded Programmierung
  - Laufzeit- und Speicher-effizient
  - Maschinennahe, Treiber-Programmierung
- C++/Java ist objektorientierte Programmiersprache
  - gut für große Anwendungen mit komplexen Strukturen
  - Objektorientierte Modellierung und Programmierung
  - GUI-Programmierung
  - Client/Server-Programmierung
  - Maschinenunabhängige Programmierung (insbesondere Java)

Unterschiede C++ vs. Java

- Java
  - durch Byte-Interpreter absolut Maschinenunabhängig
  - durch Garbage Collector einfacherer Speicherverwaltung für Applikation
  - nur Einfachvererbung
  - Byte-Interpreter ergibt Performance-Nachteile
- C++
  - eingeschränkte Maschinenunabhängigkeit (z.B. MFC-Bibliothek stark Windows-lastig)
  - Dynamische Speicherallokierung und Verwaltung durch Applikation => fehleranfällig
  - Mehrfachvererbung (kann Probleme erzeugen)
  - i.d.R. Laufzeit-optimaler als Java

Inhalt

Vom OOD zur Implementierung

- Übergang vom OO-Design zur Implementierung.
- Am Ende des objektorientierten Entwurfs existieren u.a.
  - ein Klassendiagramm,
  - ein Sequenzdiagramm und
  - ein Kommunikationsdiagramm.
- Ziel der Implementierungsphase ist es, mit Hilfe der in der OOD-Phase ermittelten Diagramme möglichst automatisch einen lauffähigen Code zu generieren.
- Anhand einer beispielhaften Anwendung soll im Folgenden die Vorgehensweise demonstriert werden.

Beispiel: Ausschnitt aus einem POS-System

- Point-Of-Sale-System (POS)
  - System zur Aufzeichnung von Verkaufsvorgängen (inkl. Zahlungen) in einem Geschäft.
  - Es besteht aus Software- und Hardwarekomponenten (Monitor, Computer, Barcodescanner, usw.).
- Ein wichtiges Szenario ist
  - der Prozess Verkauf
  - mit dem Haupt-Aktor Kassierer
- Ein Teil dieses Szenarioverlaufs dient als Ausgangspunkt für die beispielhafte Umsetzung des Entwurfs in Code.

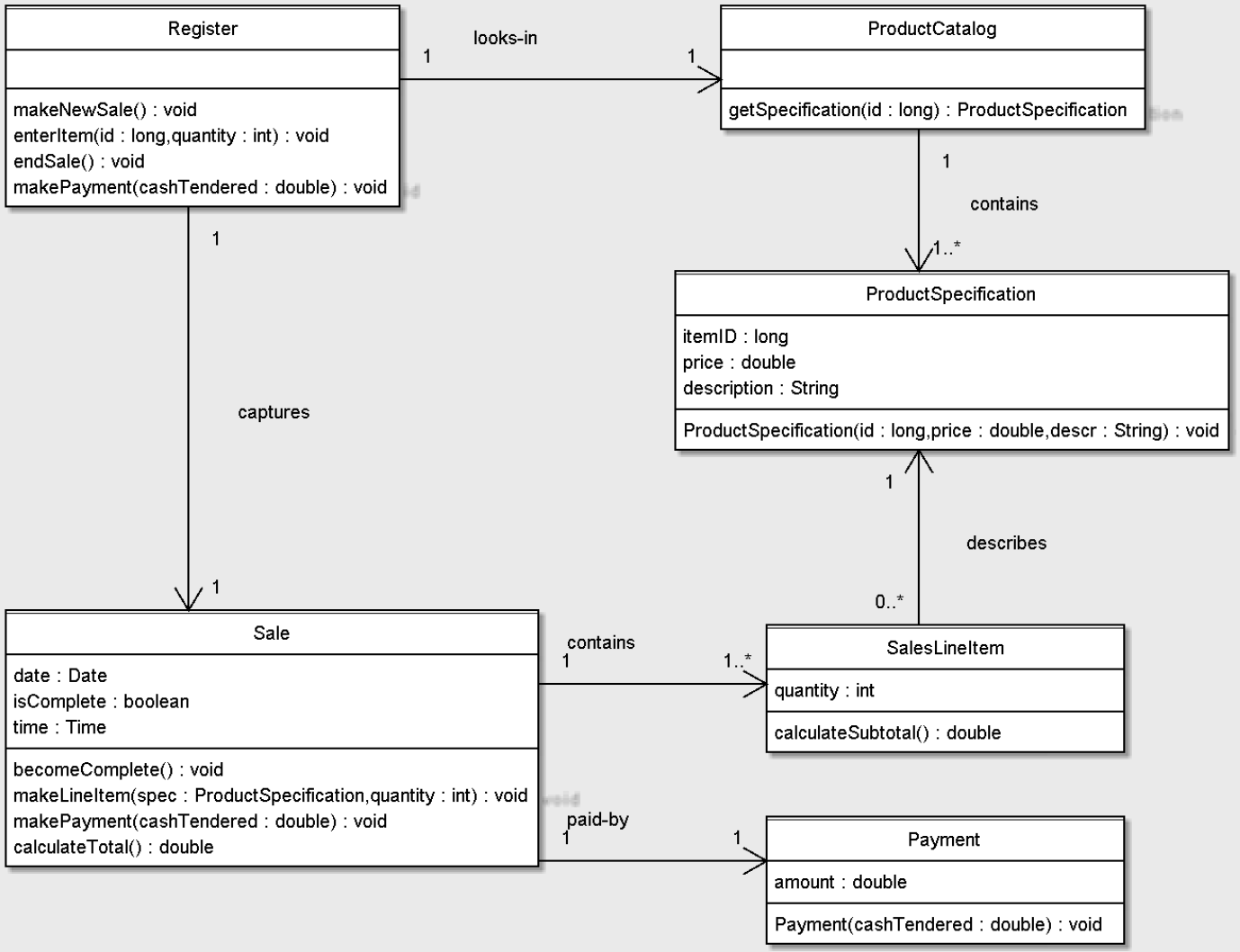
Beispiel POS-System: Hauptszenario

- Hauptszenario für den Prozess „Verkauf“:

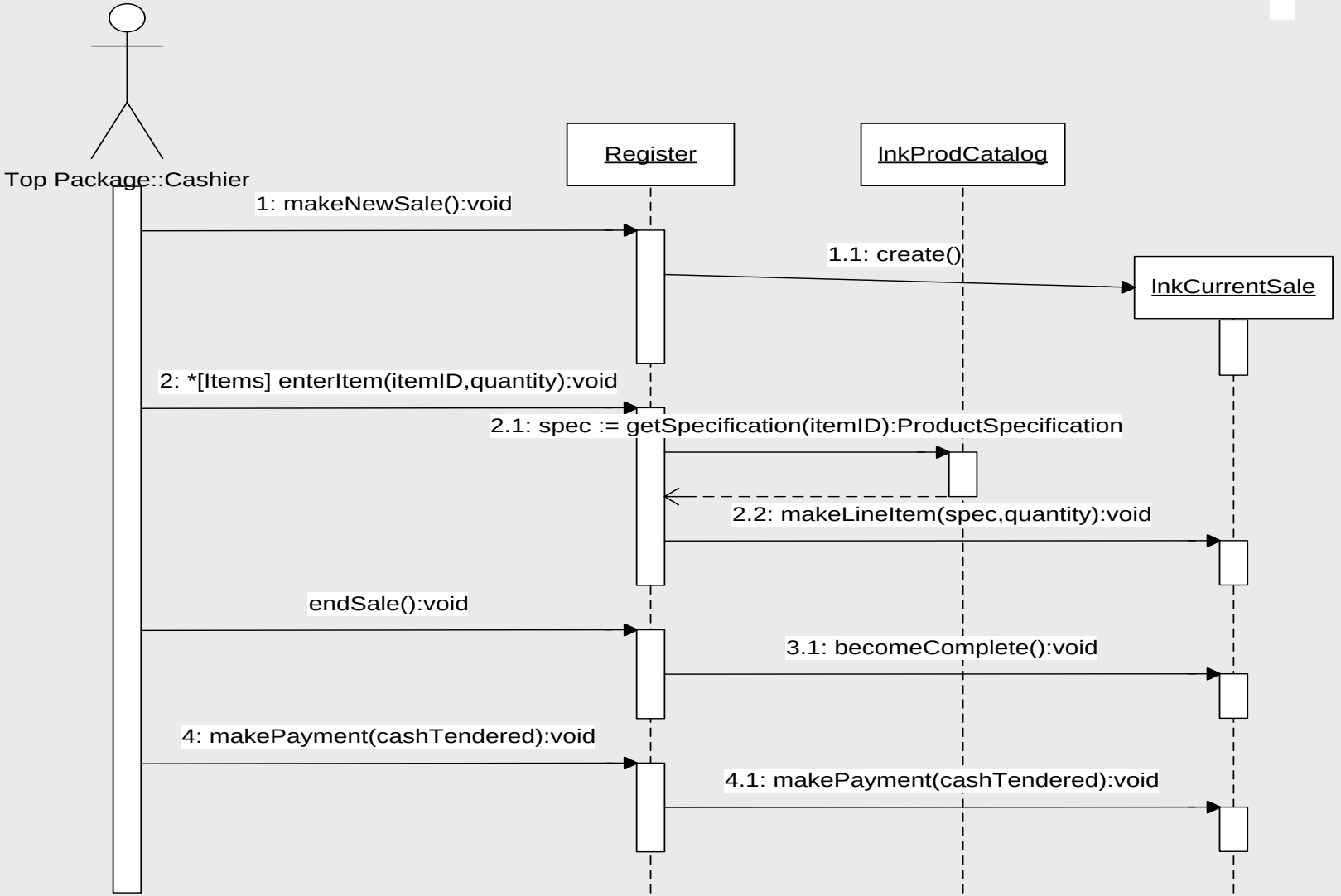
  Ein Kunde kommt mit Waren beim Kassierer an.

  1. Der Kassierer beginnt einen neuen Verkaufsvorgang.
  2. Der Kassierer gibt die Kennung des Artikels ein.
     Das System zeichnet daraufhin den Verkauf des Artikels auf und zeigt Beschreibung, Preis und Gesamtsumme an.
     Der Kassierer wiederholt den Schritt 2. solange bis alle Waren, die der Kunde kaufen will, eingegeben sind.
  3. Der Kassierer beendet den Verkaufsvorgang.
     Das System zeigt daraufhin die Gesamtsumme inklusive der enthaltenen Mehrwertsteuer an.
  4. Der Kunde bezahlt und der Kassierer gibt die Bestätigung des Vorgangs ein.
     Das System speichert den kompletten Verkaufsvorgang.
     Das System druckt einen Beleg aus.
     Der Kunde verlässt den Kassierer mit seinen gekauften Waren.

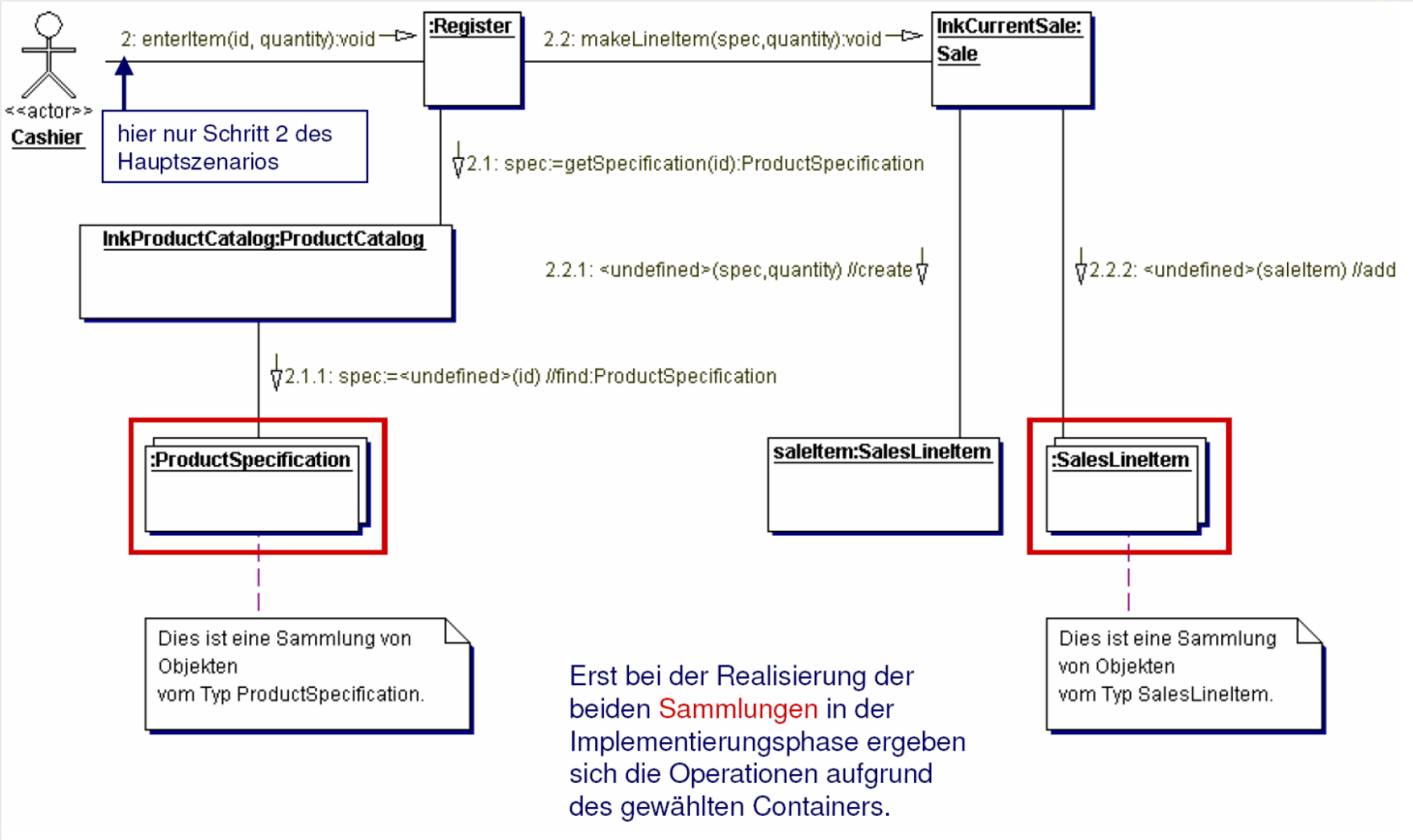
Beispiel POS-System: Klassendiagramm



Beispiel POS-System: Sequenzdiagramm (vergröbert)



Beispiel POS-System: Kommunikationsdiagramm (Ausschnitt)

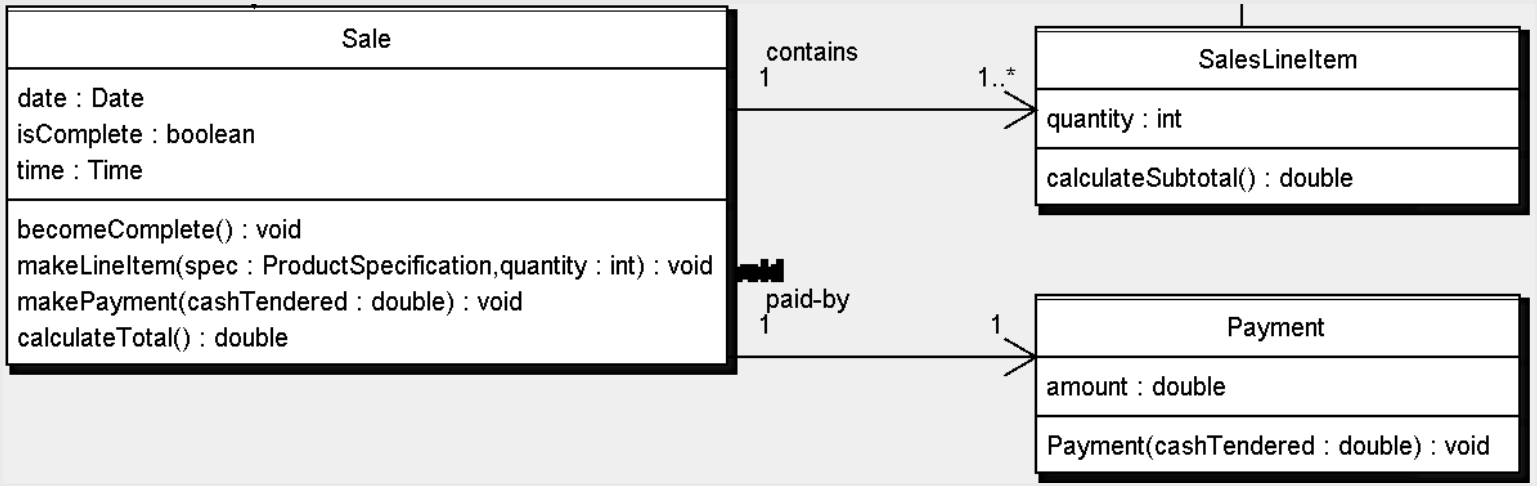


Allgemeine Vorgehensweise

1. Erzeugen der Klassendefinitionen auf Basis der Klassendiagramme.
2. Typanpassung der einfachen Attribute.
3. Typanpassung der Operationen.
4. Umsetzung/Anpassung der Assoziationen zu Referenzattributen (1:1-Beziehung) bzw. Containern/Collections (1:n-Beziehung).
5. Umsetzung der Interaktionsdiagramme in den Methodendefinitionen.
6. Fertigstellung der Methodendefinitionen (lokale Variablen, Algorithmen).

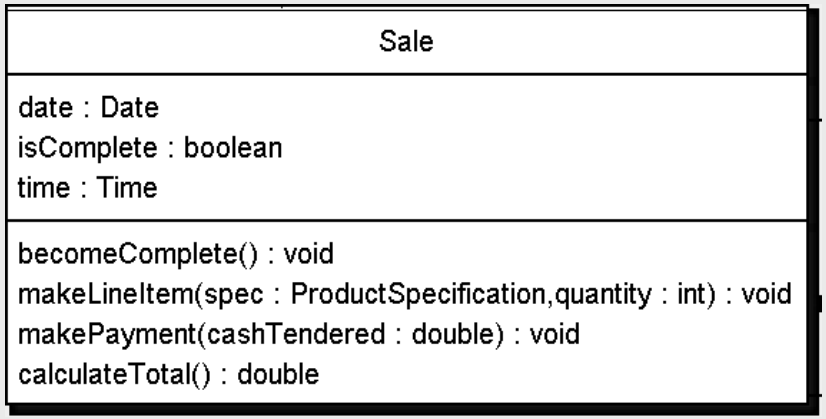
Schritt 1: Erzeugen der Klasse Sale (1)

- Verwendet wird die Klasse Sale und die assoziierten Klassen Payment und SalesLineItem.



- Die Klasse Sale wird auf Basis des Klassendiagramms in der Zielsprache Java erzeugt. Bei Verwendung eines CASE-Tools kann die Erzeugung größtenteils automatisch erfolgen.

Schritt 1: Erzeugen der Klasse Sale (2)



Anmerkung:
Source code wurde mit dem Tool "ArgoUML" generiert. Java-Import-Anweisungen wurden der Übersichtlichkeit halber gelöscht.

```
public class Sale {
  private Date date;
  private boolean isComplete;
  private Time time;

  public SalesLineItem lnkSalesItem;
  public Payment lnkPayment;

  public void becomeComplete() { }

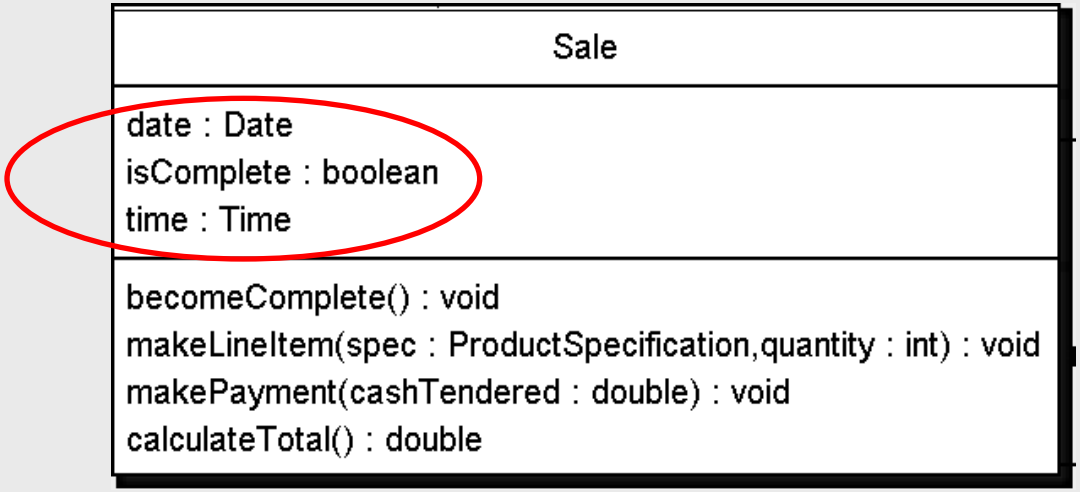
  public void makeLineItem
          (ProductSpecification spec,
           int quantity) { }

  public void makePayment(
              double cashTendered) { }

  public double calculateTotal()
    { return 0.0; }
}
```

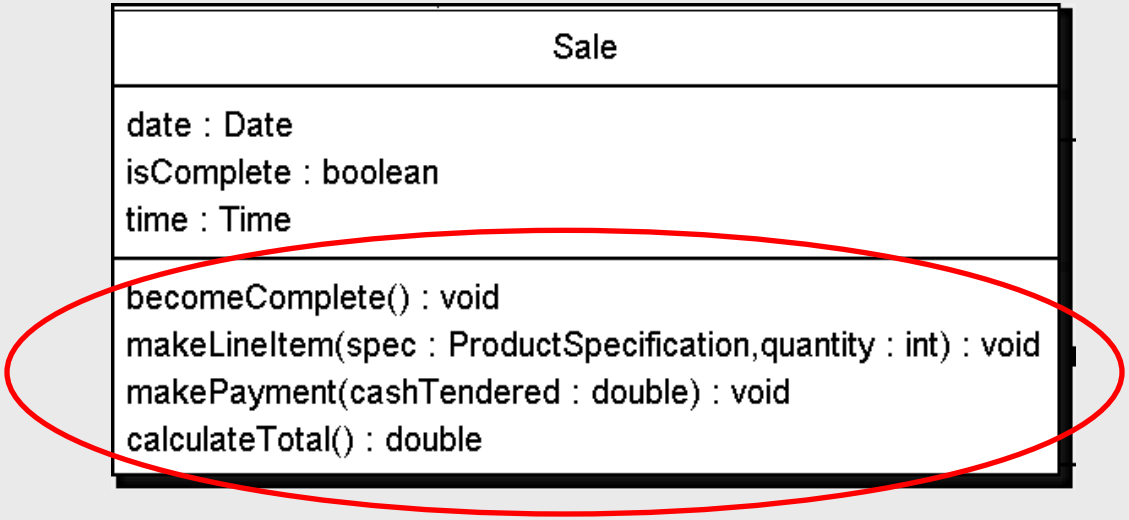
Schritt 2: Typanpassung der einfachen Attribute

- Eine manuelle Anpassung ist notwendig, falls die Datentypen der Attribute in der Zielsprache nicht vorhanden sind.
- In unserem Bsp.: in Java existiert kein Typ „Time“.
  Manuelle Anpassung notwendig: Zeiten können zusammen mit einem Datum mit Hilfe des Typs „Calendar“ verarbeitet werden.
  Alle anderen Attribute entsprechen einem Java-Datentyp.



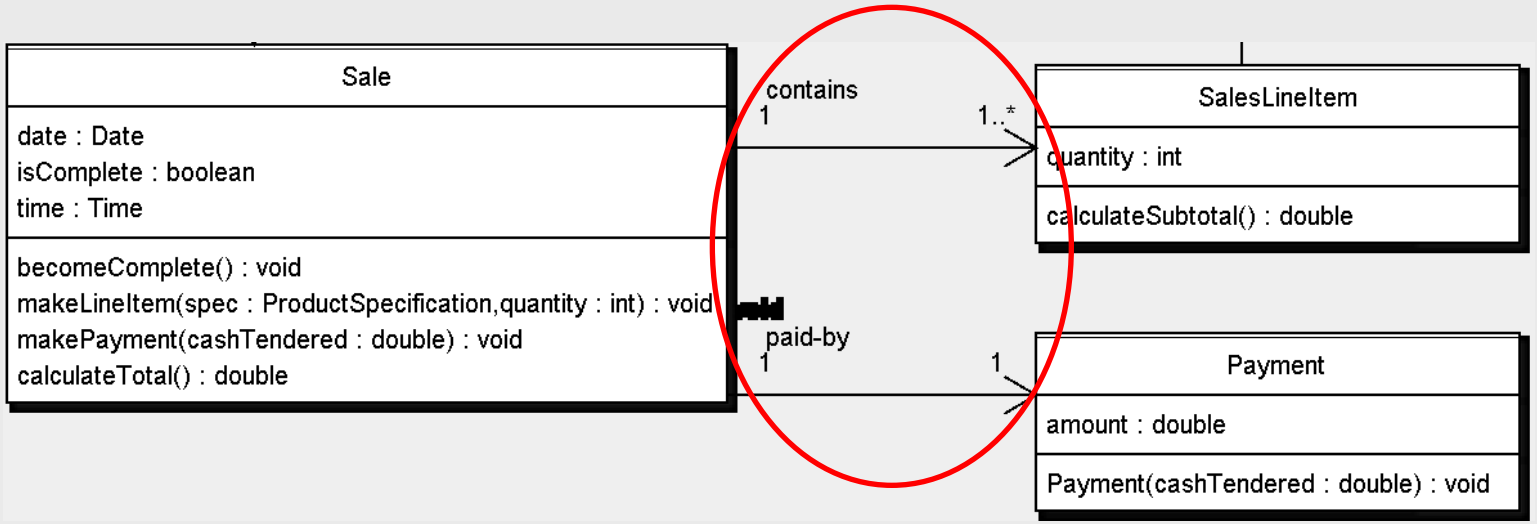
Schritt 3: Typanpassung der Operationen

- Eine manuelle Anpassung ist notwendig, falls die Datentypen der Methoden in der Zielsprache nicht vorhanden sind.
- Die Methoden der Klasse Sale verwenden ausschließlich existierende Java-Datentypen bzw. selbst definierte Klassen (ProductSpecification). Daher ist in diesem Fall **keine** manuelle Anpassung notwendig.



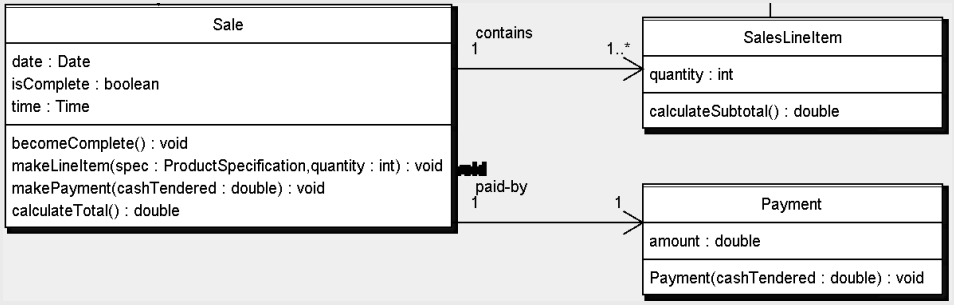
Schritt 4: Umsetzung der Assoziationen (1)

- Eine manuelle Anpassung ist notwendig, falls die Datentypen der Referenz-Attribute (ergeben sich aufgrund der Assoziationen) in der Zielsprache nicht vorhanden sind.
- Die korrekte Umsetzung von 1:n-Beziehungen durch passende Daten-Container (z.B. Listen, Mengen, Arrays) ist zu prüfen.



Schritt 4: Umsetzung der Assoziationen (2)

- 1:1-Beziehung zwischen „Sale“ und „Payment“ (Assoziation „Paid-by“)
- Aufgrund der Richtung und der Multiplizität der Assoziation ergibt sich, dass jedes Objekt der Klasse „Sale“ ein Objekt der Klasse „Payment‘ referenzieren muss. Dies wird vom UML-Tool ArgoUML durch Erzeugung des Attributes lnkPayment automatisch umgesetzt.

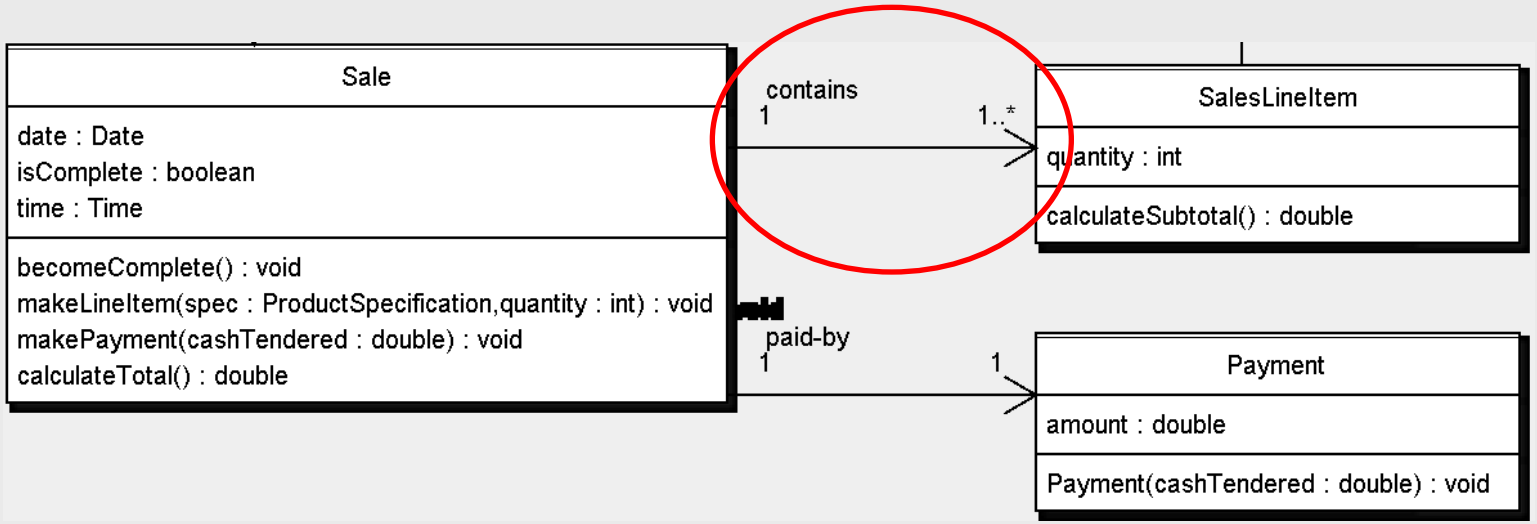



```
import java.util.Calendar;

public class Sale {
...
  private Calendar dateTime
                    = new Calendar();
  private boolean isComplete = false;
...
  private Payment lnkPayment;
...
}
```

Schritt 4: Umsetzung der Assoziationen (3)

- 1:n-Beziehung zwischen "Sale" und "SalesLineItem" (Assoziation „contains“)
- Aufgrund der Richtung und der Multiplizität dieser Assoziation ergibt sich, dass jedes Objekt der Klasse "Sale" i.A. mehrere Objekte der Klasse "SalesLineItem" referenzieren muss. In diesem Fall wird die Umsetzung nicht automatisch vorgenommen, sondern muss manuell durch Festlegung einer Sammlung (hier: ArrayList) erfolgen.



Schritt 4: Umsetzung der Assoziationen (4)

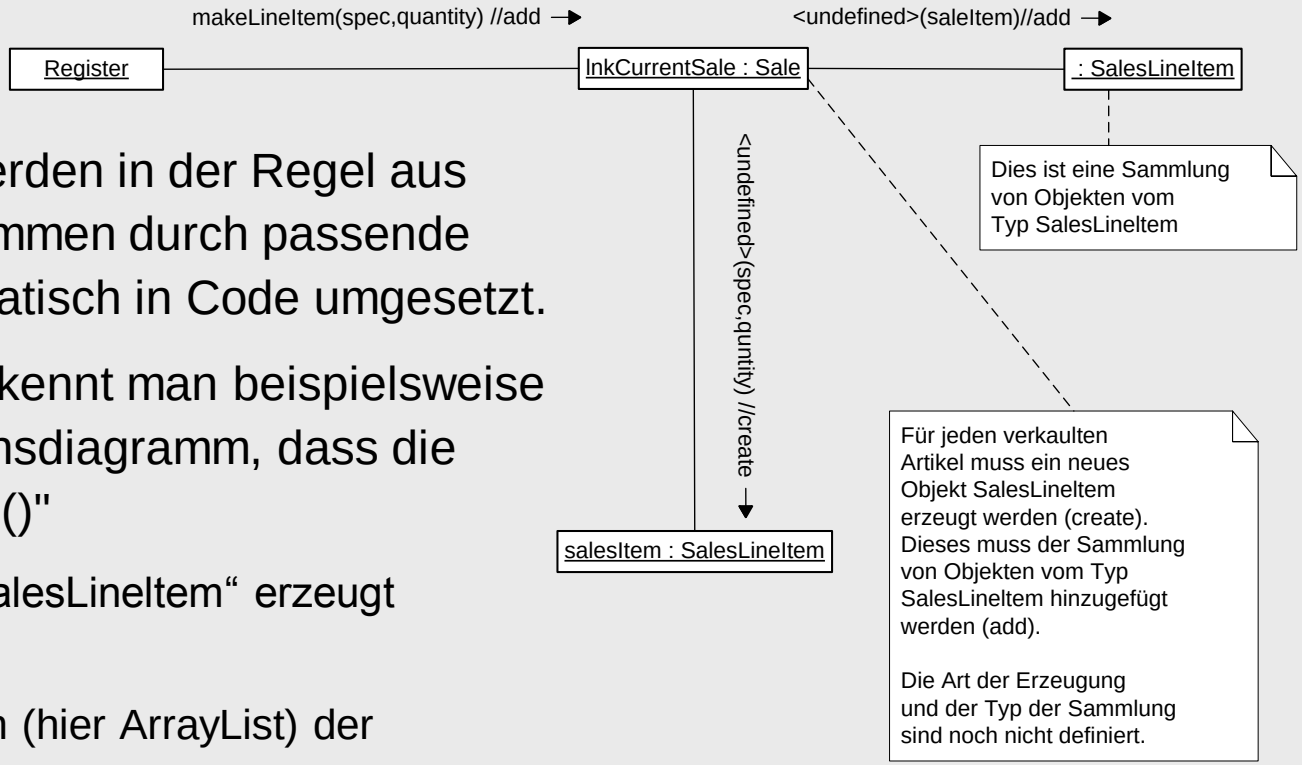
```
import java.util.Calendar;

public class Sale {
...
  private Calendar dateTime
                    = new Calendar();
  private boolean isComplete = false;
...
  private Payment lnkPayment;

  private ArrayList<SalesLineItem> lnkSalesLineItems
          = new ArrayList<SalesLineItem> ();
...
}
```

- Anmerkung:
  das Tool "ArgoUML" erzeugt hier den Java-Datentyp "Vector", der allerdings nicht "von Haus aus" typenrein ist. Mit der "ArrayList"-Collection wird sichergestellt, dass nur Objekte vom Typ "SalesLineItem" in die Collection aufgenommen werden.

Schritt 5: Umsetzung der Interaktionsdiagramme (1)



- Die Aufrufsequenzen werden in der Regel aus den Interaktions-Diagrammen durch passende Methodenaufrufe automatisch in Code umgesetzt.
- Für das POS-System erkennt man beispielsweise aus dem Kommunikationsdiagramm, dass die Methode "makeLineItem()"
  1. ein neues Objekt „SalesLineItem“ erzeugt (**create**) und
  2. dieses der Collection (hier ArrayList) der "SalesLineItem"-Objekte hinzufügt (**add**).
- Im Kommunikationsdiagramm ist nicht festgelegt («**undefined**»), wie die Erzeugung erfolgt und welche Art der Collection zu verwenden ist.

Schritt 5: Umsetzung der Interaktionsdiagramme (2)

- Definition der Klasse "Sale"
  - Methode "makeLineItem erzeugt ein "SalesLineItem und fügt das neue Objekt der ArrayList "lnkSalesLineItems hinzu

```
import java.util.Calendar;

public class Sale {
...
  private ArrayList<SalesLineItem> lnkSalesLineItems
          = new ArrayList<SalesLineItem> ();
...
  public void makeLineItem (ProductSpecification spec, int quantity )
  {
    SalesLineItem item = new SalesLineItem (spec, quantity);
    lnkSalesLineItems.add (item);
  }
  ...
}
```

Schritt 6: Fertigstellung der Methodendefinitionen

- Die weitere Programmiertätigkeit befasst sich mit der Vervollständigung des Programmcodes.
- In der Regel sind jetzt noch die Methodenrümpfe manuell zu programmieren
- Im obigen Beispiel aus Schritt 5 ist dies nicht notwendig, da die Methoden **new** und **add** in Java bereits definiert sind.

Round-Trip-Engineering

- Forward Engineering
  - Fertiges Softwaresystem ist Ergebnis des Entwicklungsprozesses
  - Erzeugung des Programmcodes aus Entwurfsmodellen
- Reverse Engineering
  - vorhandenes Softwaresystem ist Ausgangspunkt der Analyse
  - Erzeugung des Entwurfsmodells aus Programmcode
- Round-Trip Engineering
  - Kombination aus Forward Engineering und Reverse Engineering

Inhalt

Fehleranfällige Konstrukte

- Jede Programmiersprache (z.B. C, C++, Ada, Java) enthält fehleranfällige Konstrukte.
- Ein Beispiel zur Programmiersprache C:
  - Gemäß dem ISO-C-Standard (Anhang G) ist die Auswertungsreihenfolge von Ausdrücken nicht spezifiziert.
  - Folgender Code sollte also nicht geschrieben werden:

```
...
int x = 1;
int ar[] = {0, 1, 2};
int y = ar[++x] + x;
...
```

  - bei Auswertung von links nach rechts hat y den Wert 4, bei Auswertung von rechts nach links hat y den Wert 3

Übung

Übungsaufgabe MISRA-Regeln und

Beschreibung:

Es gibt in C/C++ zahlreiche Schwachstellen, die ungewollt zu Fehlern in Programmen führen können. Die Anwendung von Codierregeln wie MISRA reduziert das Fehlerrisiko. Desgleichen können Werkzeuge wie PC-Lint Schwachstellen im C/C++-Code aufdecken

Aufgabe: Analyse und Präsentation von MISRA – Regeln,
Analyse und Präsentation eine Bug of the Month
der Webseite der Fa. Gimpel (www.gimpel.com)

Genaue Aufgabenstellung s. Aufgabenblatt

Safe Subsets

- Zur Vermeidung fehleranfälliger Konstrukte, insbesondere für sicherheitskritische Software, werden Regeln definiert, die den ursprünglichen Wortschatz einschränken.
- z.B. in Bezug auf die Auswertungsreihenfolge von Ausdrücken:
  - „Verwende den Inkrement-Operator immer in einem eigenen separaten Ausdruck.“
  - Mögliche Lösung zum Problembeispiel:

```
...
++x;
int y = ar[x] + x;
...
```

- Auf diese Weise entstehen sogenannte „Safe Subsets“
  - z.B. MISRA-C für C

Allgemeine Codierungsregeln

- Verständlichkeit und Lesbarkeit für alle Betroffenen sind wichtiger als ein geringerer Schreibaufwand:
  - gute Kommentierung
  - problemorientierter Aufbau
  - angemessene Datenstrukturen
  - einheitlicher Programmierstil, möglichst unternehmensspezifisch, zum Teil projektspezifisch

- Verständlichkeit

  ist wichtiger als Betriebsmittelbedarf (Speicherplatz, Rechenzeit), insbesondere
  - Lesbarkeit vor Zeitersparnis
  - Klarheit vor Genialität
    (Programmpflege, Nachweisführung!)
  - gilt nur eingeschränkt im Embedded-Bereich !

Allgemeine Codierungsregeln

- Problemorientierter Aufbau

  Programm soll in

  - Daten- und
  - Ablaufstruktur

  das Problem widerspiegeln

  - Verwendung von Sprachelementen, die die Absicht des Programmierers wiedergeben

- Stil-Einheit in Großprojekt
  - Einheitlicher Programmierstil
  - Einheitlicher Dokumentationsstil
  - Einheitliches Programmlayout, z.B.
    - Einrückungen logischer Strukturen
    - Zeilenaufteilung: nur eine Anweisung pro Zeile

Beispiel: Änderung von Programmbefehlen

- Vermeiden von Tricks, etwa
  - Mehrfachverwendung von Speicherplatz für verschiedene Zwecke
  - Veränderung des Wertes des Schleifenzählers im Schleifenrumpf
  - Änderungen von Befehlen durch das Programm selbst

- Beispiel: Ruby-Programm

  Bei der Ausführung der Methode greet() wird diese Methode durch „def self.greet“ für das eigene Objekt neu definiert.

  Es werden 2 Objekte (hw, hw2) erzeugt.
  Die erzeugte Ausgabe ist:

```
Hallo, Welt!
Tschüss, Welt!
Hallo, Welt!
```

```
class HelloWorld
  def greet()
    print("Hallo, Welt!\n");
    def self.greet()
      print("Tschüss, Welt!\n")
    end
  end
end

hw = HelloWorld.new
hw2 = HelloWorld.new
hw.greet()
hw.greet()
hw2.greet()
```

Codierungsregeln

- Bezeichner
  - eindeutige Identifizierung der Objekte,
  - leicht erkennbare Bedeutung
  - Begriffliche Bedeutung: “So lang wie nötig, so kurz wie möglich!“ “konsistente“ und “bedeutungsreiche“ Variablen-Namen
- Deklaration und Initialisierung
  - alle Variablen möglichst bei der Deklaration initialisieren
- Keine Zahlen direkt im Code verwenden
  - stattdessen Konstanten definieren (Programmpflege!)

Inhalt

Exkurs: Codierungsregeln lt. MISRA

MISRA C

Der Safety Coding Standard

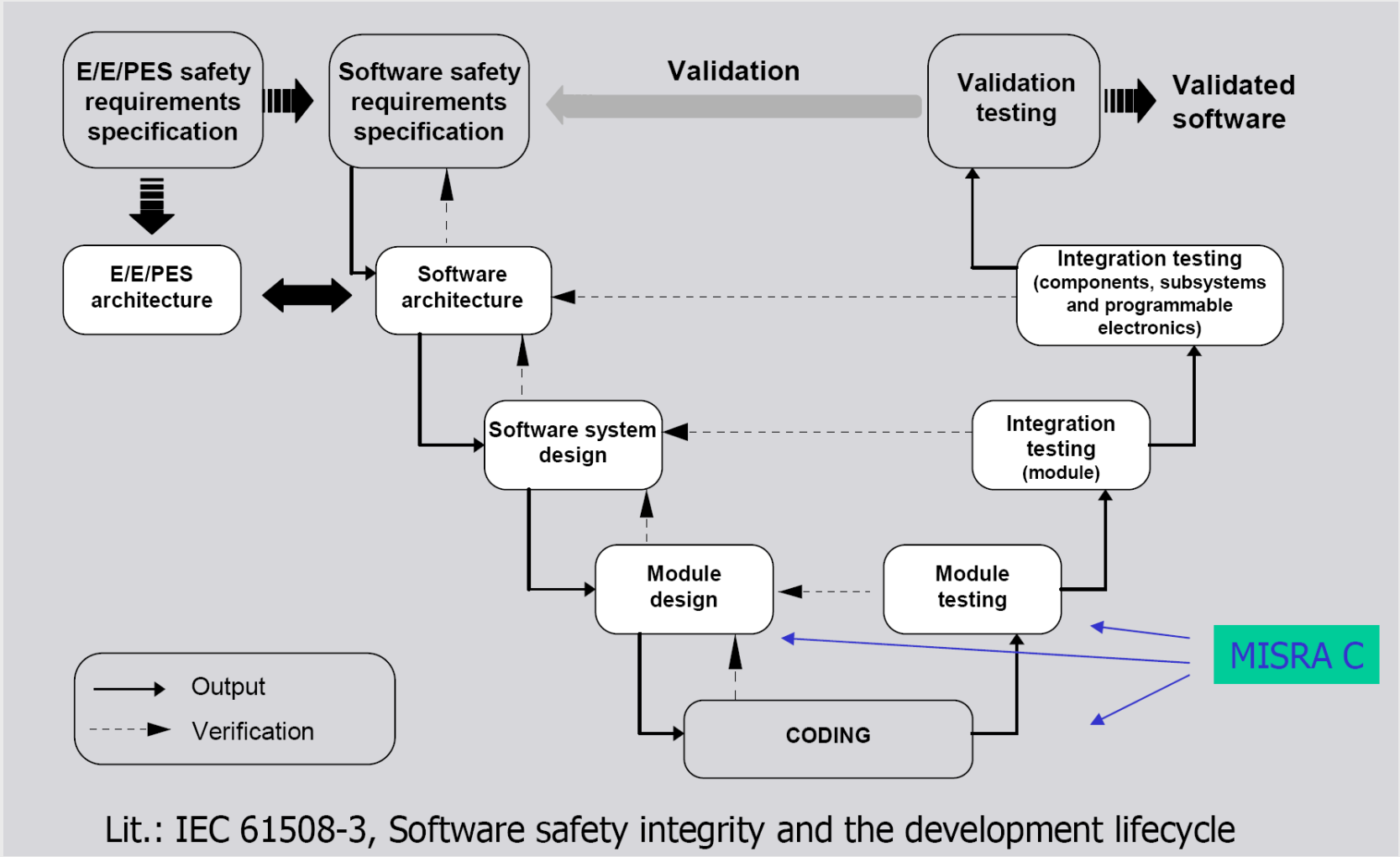
für die Programmiersprache C

Exkurs: Codierungsregeln lt. MISRA

- Ein übersetzbares, aber nicht verstehbares C Programm:
  (wird auch als "source code obfuscation" bezeichnet)

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')#
 \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-
 O;m .vpbks,fxntdCeghiry"),a+1);}1 ) }
```

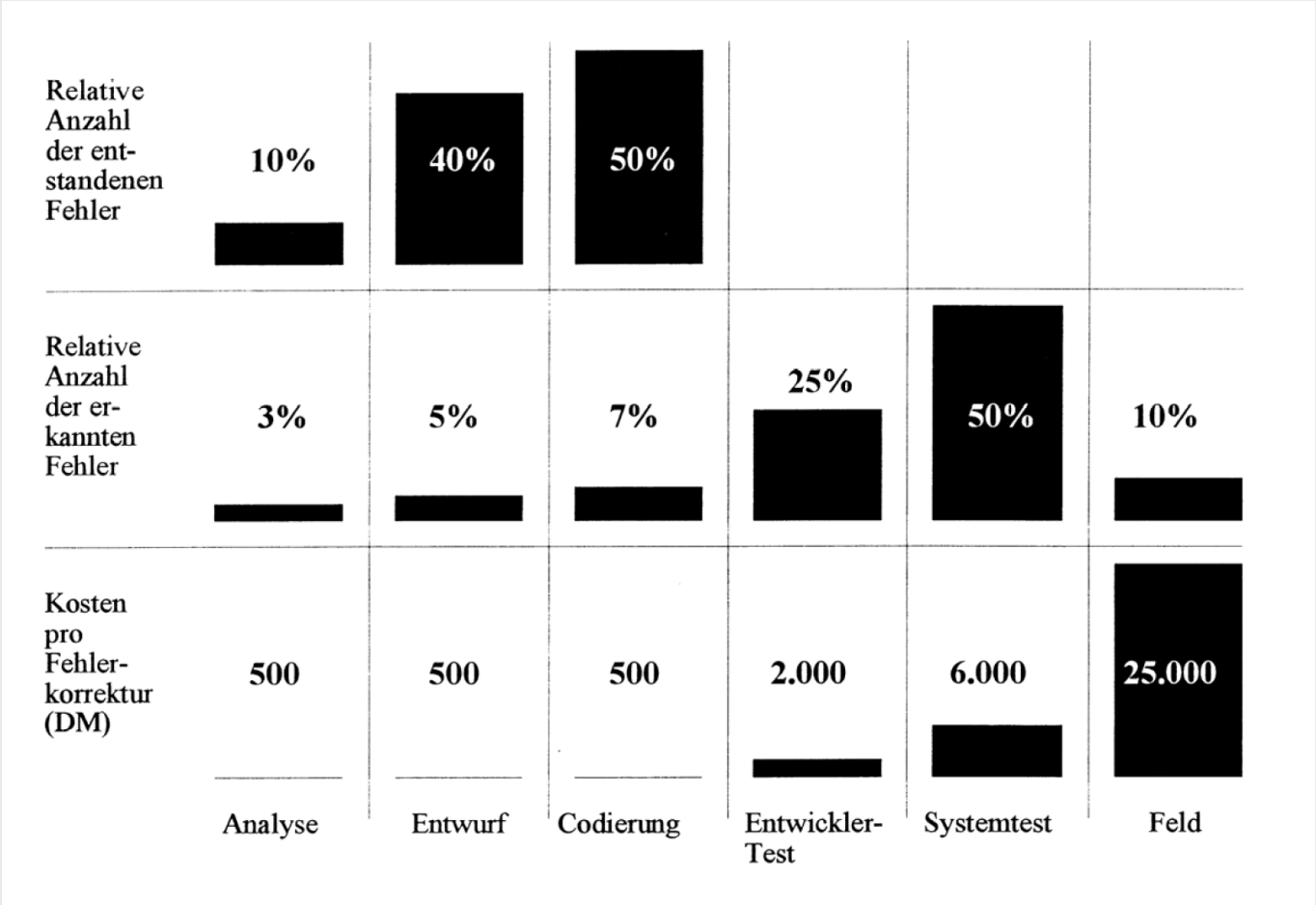
Bereich des V-Zyklus



Lit.: IEC 61508-3, Software safety integrity and the development lifecycle

Fehlerursachen und Fehlerkosten

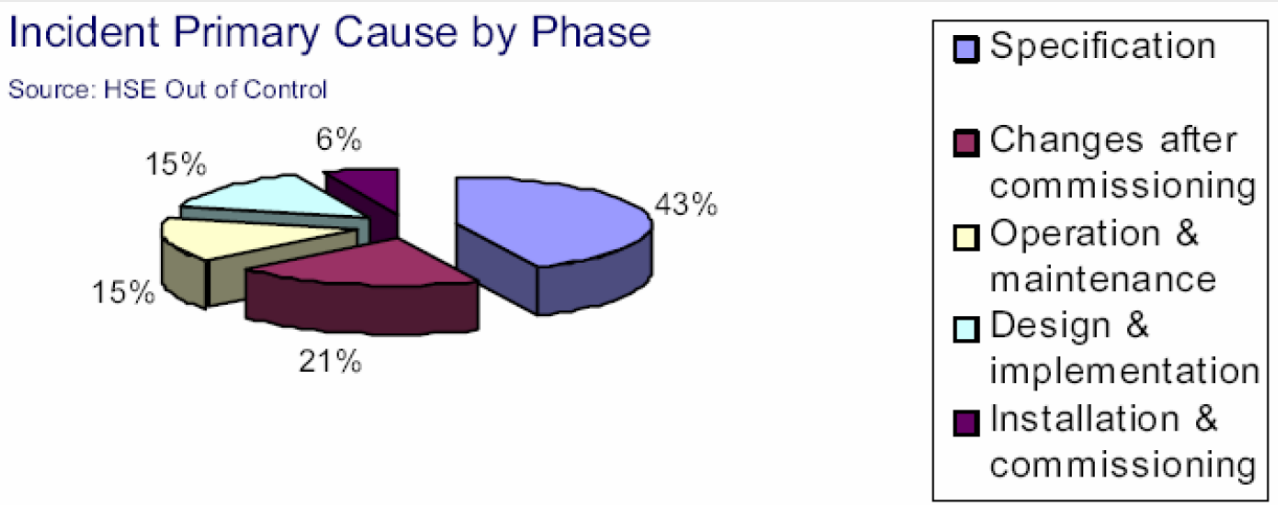
- Empirische Ergebnisse (Quelle: Liggesmeier)



Sichereres C mit MISRA

- **MISRA**: Motor Industry Software Reliability Association:
  - Ein Zusammenschluss von Fahrzeug-Herstellern, Zulieferern und Ingenieurs-Beratungs-Firmen von Großbritannien
- Historie:
  - MISRA-C: 1998[1]: Erstveröffentlichung April 1998, Nuneaton
  - MISRA-C: 2004[2]: veröffentlicht October 2004, Nuneaton
  - MISRA, Development Guidelines for Vehicle Based Software [6], November 1994, Nuneaton
  - ...
- Siehe auch: http://www.misra.org.uk/
- http://computing.unn.ac.uk/staff/cgam1/teaching/0703/misra%20rules.pdf

MISRA-C kann bei Reduzierung einiger Fehlertypen unterstützen



- Can help to reduce the 15% of defects that are introduced during design and coding
- Embodies accumulated automotive software engineering knowledge and best practice
- A suitable subset of C for use at SIL2 and above, as suggested by the MISRA Guidelines
- Must be used in conjunction with a structured development process
- Use of MISRA C in itself is no guarantee of software quality – nearly two thirds of defects arise from errors in specification or errors made during changes to specification from which MISRA C can offer little protection

Source: S. Montgomery, The Role of MISRA C in Developing Automotive Software, Ricardo UK

Hintergrund: Benutzung von C in der Industrie

- Erhöhte Wichtigkeit der C Programmiersprache (in der Automotive-Industrie)
- Potential für Portierungsfähigkeit auf große Anzahl verschiedener Micro-Prozessoren
- C ermöglicht high-level, low-level, input/output Operationen
- Erhöhte Komplexität von Applikationen zwingt zum Gebrauch einer Hochsprache gegenüber Assembler-Sprachen
- C kann kleineren und weniger Speicherplatz verbrauchenden Code erzeugen als viele andere Hochsprachen (C++, Java)
- Vermehrter Gebrauch von Code-Generatoren aus Modellierungs-Tools

Unsicherheiten in der Sprache C

Ursachen für nicht beabsichtigtes Verhalten des C-Codes:

- Der Programmierer macht Fehler
  - in Bezug auf Stil und Ausdruck
  - C-Syntax (Tippfehler wie "=" anstatt "==")
  - die Philosophie von C ist, dass der Programmierer weiß, was er tut (keine PASCAL-ähnliche Typsicherheit)
- Der Programmierer missversteht die Programmiersprache (z.B. Operator-Präzedenz)
- Der Compiler macht nicht das, was der Programmierer erwartet
- Der Compiler hat Fehler
- Laufzeit-Fehler: C macht keine Laufzeit-Checks für z.B. Overflows, gültige Address-Zeiger, Array-Out-of-bounds, ...

MISRA-C: Anpassung einer Untermenge von Regeln

- Abweichungen müssen beurteilt werden nach Kriterien der Notwendigkeit und Sicherheit (Safety).
- Der Abweichungs-Prozess sollte Teil des formellen Qualitäts-Managements ein.
- Zwei Kategorien von Abweichungen:
  - Projekt-Abweichungen:
    - erlaubte Aufweichung von Regel-Anforderungen unter speziellen Umständen
    - sollte gereviewed werden als Teil des formellen Abweichungs-Prozesses
  - Spezifische Abweichungen
    - Abweichung einer bestimmten Regel in einer einzelnen Datei
    - Wird gereviewed im Entwicklungs-Prozess
- Der Abweichungs-Prozess sollte nicht benutzt werden, um die MISRA-Absichten zu untergraben

MISRA-C: Kategorisierung der Regeln

Regel	Kategorie	Regel	Kategorie
1	Umgebung	12	Ausdrücke
2	Spracherweiterungen	13	Kontrollfluß-Anweisungen
3	Dokumentation	14	Kontrollfluss
4	Zeichensatz	15	Switch-Ausdrücke
5	Identifiers	16	Funktionen
6	Datentypen	17	Zeiger und Arrays
7	Konstanten	18	Strukturen und Unions
8	Deklarationen und Definitionen	19	Präprozessor-Anweisungen
9	Initialisierungen	20	Standard-Bibliotheken
10	Arithmetische Typumwandlungen	21	Laufzeit-Fehler
11	Zweiertyp-Umwandlungen		

Beispiele für MISRA-Regeln

Regel 1.4 (Forderung):

- Der Compiler/Linker muß garantieren, dass eine 31-Zeichen-Unterscheidung sowie Groß-/Kleinschreibung bei externen Identifiers unterstützt wird.

Regel 1.5 (Ratschlag):

- Gleitkomma-Implementierungen sollten sich nach dem Gleitkomma-Standard richten
- z.B.:

```
/* IEEE 754 single-precision floating point */
typedef float float_32t
```

Beispiele für MISRA-Regeln

Regel 2.1 (Forderung):

- Assembler-Aufrufe sollen eingekapselt und isoliert sein.
- Beispiel:

```
#define NOP asm("   NOP")
```

Regel 2.2 (Forderung):

- Sourcecode soll ausschliesslich Kommentare der Form /* ... */ benutzen

Regel 2.3 (Forderung):

- Die Zeichenfolge /* soll nicht in einem Kommentar benutzt werden

Beispiele für MISRA-Regeln

Regel 12.1 (Ratschlag):

- Es sollte möglichst wenig von den C-Regeln für Operator-Präzedenzen abhängig gemacht werden.

Regel 12.2 (Forderung):

- Das Resultat eines Ausdrucks soll immer gleich sein bei jeder möglichen Reihenfolge der Abarbeitung

Regel 12.3 (Forderung):

- Der sizeof-Operator soll nicht in Ausdrücken benutzt werden, die Seiteneffekte beeinhalten
  Beispiel:
  - ```
    int32_t i; int32_t j;
    j = sizeof (i=1234);
    /* j is set to the sizeof the type of i which is an int */
    /* i is not set to 1234.
    ```
    ```

Beispiele für MISRA-Regeln

Regel 12.4 (Forderung):

- der rechte Operand eines logischen && oder || Operators soll keine Seiteneffekte beinhalten

Regel 12.6 (Forderung):

- Die Operanden von logischen Ausdrücken (&&, || und !) sollen Bool'sche Ausdrücke sein. Bool'sche Ausdrücke sollten nirgends sonst als Operanden benutzt werden.

Regel 12.7 (Forderung):

- Bit-Operationen sollen nicht auf Operanden angewandt werden, deren Datentyp vorzeichenbehaftet ist

Tools für Überprüfung der MISRA-Regeln

- **PCLint (Fa. Gimpel Software)**
  - PC-Lint überprüft C/C++ Quellcode. Im Gegensatz zu einem Compiler analysiert PC-Lint C/C++ Quellen modulübergreifend.
  - PC-Lint für C/C++ kontrolliert die Einhaltung der MISRA Richtlinien.
  - PC-Lint überprüft Quellcode auf typische C++- und C-Fehler und findet:
    - nicht initialisierte Variablen
    - vererbte, nicht virtuelle Destruktoren
    - Typ-Unverträglichkeiten
    - falsch formulierte Macros
    - unbeabsichtigtes Name-hiding
    - Statische Variablen in In-line-Funktionen von Headern
    - Fehler beim Kopieren einer Basisklasse oder bei der Benutzung des Copyconstructors der Basisklasse
    - Gebrauch von 'throw'in einem Destruktor
    - unreferenzierte 'catch'-Parameter
    - virtuelle Funktionen mit einem default Parameter
    - ...

Tools für Überprüfung der MISRA-Regeln

- **QA-C/MISRA (Fa. QA Systems)**
  - Lokalisiert Quellcode, der nicht den MISRA-Regeln entspricht
  - Verbindet Meldungen direkt mit dem Quellcode und den entsprechenden MISRA-Regeln
  - Individuell auf jedes MISRA Subset konfigurierbar (auch nur für Teile eines Projektes)
  - Querverweise (HTML) zu Regeldefinitionen und erklärenden Beispielen
  - Reports über die Softwarequalität (Art und Häufigkeit von Regelverletzungen etc.)
  - Erzeugt textliche und grafische Metriken (Testbarkeit, Pflegbarkeit, Portabilität)
  - Unterstützt Qualitätsinitiativen wie CMM, ISO9003/EN29003, ISO 9126, IEC 61508, IEC 26262, DO-178B, Def Stan 00-55
  - Integrationen zu Entwicklungsumgebungen wie z.B. Visual Studio oder Codewright und allen gängigen Versionskontrollsystemen

Zusammenfassung

MISRA-C definiert ein sichere Untermenge von C, aber

- wie erreicht man mehr Sicherheit ("Safety") in einem technischen System, dass aus Software, Hardware und Mechanik besteht ?
- wie soll man mit MISRA ein bestimmtes Integrity Level ("IL") erreichen ?
- wie soll man mit MISRA ein bestimmtes Safety Integrity Level ("SIL") erreichen ?

Die folgenden Standards sollen helfen:

- IEC 61508 (Definition der Safety Integrity Levels)
- IEC 26262 (Anpassung der 61508 an Automotive Erfodernisse)
- MISRA, Development Guideline for Vehicle Based Software
- ISO 26262 (Anpassung der IEC 61508 an Automotive)

Zum Schluss dieses Abschnitts …

Noch Fragen ??