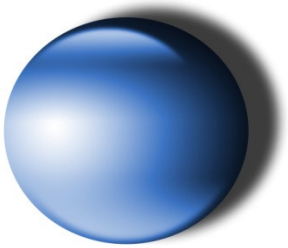
Software Engineering

RWTH AACHEN



Einführung in die Softwaretechnik

Vorlesung, Teil 2

Kapitel 7. Imlementierung

Prof. Dr. Bernhard Rumpe
Lehrstuhl Informatik 3 (Software Engineering)
Rheinisch-Westfälische Technische Hochschule Aachen

http://www.se-rwth.de/

Definition: Implementierung

- Definition: Implementierung ist die Menge aller Programmier-Aktivitäten.

- Die Implementierung geht von einer gegebenen System-Architektur und detaillierten Spezifikation der Funktionalität aus.
- Ihre Ergebnisse sind:
  - Quellprogramm einschließlich integrierter Dokumentation
  - Lauffähiges System
  - Testplanung und Testdokumentation

- Die Implementierungs-Aktivitäten sind daher stark mit den Test-Aktivitäten verzahnt, um die Qualität des Systems sicher zu stellen.
- Varianten der „Implementierung“:
  - Legacy-System ist zu erweitern oder anzupassen
  - Rapid Prototyping: auf Design und Detailspezifikation wird dabei u.U. verzichtet

7. Implementierung
7.1. Auswahl der Implementierungssprache

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Objektorientierte Programmierung

- Beispiel: Java (C++, Oberon, Modula-3, Smalltalk, C#, ...)
- Die Merkmale der OO:
  - Objekte (Klassen) zur Kapslung von Daten und Funktionen
  - Objekte dynamisch erzeugen: Objektidentität
  - Vererbung (in ihren verschiedenen Ausprägungen)
- OO Sprachen subsumieren prozedurale Programmierung
- Es erfordert aber eine wesentlich andere Vorgehensweise, um Vorteile der OO zu nutzen:
  - Vererbung als Mechanismus zur Anpassung und Verbesserung der Wiederverwendung
  - „Gutes Design“, um Wartbarkeit und Erweiterbarkeit zu stützen
  - Codingstandards für Lesbarkeit

- Fazit:
  - OO ist heute das Mittel der Wahl für große Projekte, effizienter geht es aber oft mit Spezialsprachen

Prozedurale Programmiersprachen

- Beispiele: Modula-2, (Pascal, C, Fortran)
- Ideen zum Modul-Konzept teilweise vorhanden
- Komfortable Definition von Datenstrukturen im Speicher
- Seiteneffekte
- Trennung von Datenstruktur und Funktionen macht Wartbarkeit schwieriger

- Fazit:
  - Für kleinere und mittlere Systeme geeignet
  - Prozedurale Programmierung ist in der objektorientierten Programmierung subsumiert und wird deshalb kaum mehr in Reinform verwendet

Funktionale Programmiersprachen

- Beispiele: Gofer, Haskell, ML
- Seiteneffektfrei und dadurch leicht verstehbar
- Sehr mächtiges Typsystem (bei den Beispielen)
- Patternmatching auf Argumenten
- Effiziente Definition von Datenstrukturen und Funktionen
  - `data Tree = Leaf(Int) | Node(Tree,Int,Tree)`
- Funktionen höherer Ordnung (Funktionen über Funktionen)
  - `twice f x = f(f(x))`
- Kompakte Formulierung
- Fazit:
  - Effektive Programmierung aber langsamere Ausführungszeiten.
  - Geeignet für schnelle Hacks, skaliert nicht für große Programmsysteme
  - Schwierigkeiten mit interaktiven Systemen (z.B. GUI)

Spezialsprachen

- Logikprogramme: Prolog
  - Logische Aussagen in Hornklauselform als Programm
- Visuelle Programmierung
  - a) Komposition des Programms aus Bausteinen
  - b) Modellierung z.B. mit ausführbaren Statecharts
- Parallele Programmiersprachen
  - für massiv verteilte Systeme
- Skriptsprachen
  - Beispiel: Python
  - Meist kein festes Typsystem
  - effektive Module zur Textbearbeitung (z.B. reguläre Ausdrücke)
  - Neu: immer bessere Integration mit anderen Sprachen (z.B. Python + Java)
- Weitere Spezialsprachen: HTML, JSP, XML, SQL, PHP, ...

Weitere Auswahlkriterien

- Welche technische Umgebung ist gegeben?
  - Legacy-System? Gibt es eine Vorgabe des Unternehmens?
- Human Factor: Welche Kenntnisse/Vorlieben besitzen die Programmierer?
- Welche Bibliotheksfunktionen werden benötigt?
- Wie gut ist die Werkzeugunterstützung?
  - Compiler, Debugger, IDE, ...

Beobachtungen - 1

- Fehlende Spracheigenschaften werden durch standardisierte Klassenbibliotheken abgedeckt:
  - I/O wurde in C als erstes durch Bibliotheksfunktionen standardisiert
  - Threads/Nebenläufigkeit wird in Java über die Bibliothek realisiert
  - Kommunikation, Datenspeicherung wird nicht über Sprachprimitive, sondern über Bibliotheksfunktionen angeboten (Middleware)
  - Security (z.B. Java Sandbox oder Verschlüsselung)
  - Komponentenkonzepte via Bibliothek (EJBs)

Beobachtungen - 2

- Die Kooperationsfähigkeiten der Sprachen erlauben die Anwendung der jeweils besten Sprache:
  - Corba, .NET, Embedded SQL (z.B. in Java)
  - Java Server Pages (JSP) = Java + HTML
  - Modul-Import über API's (Python in Java)
  - Generatoren wie Yacc: für Grammatik -> Parser

- Werkzeuge für das Management der Implementierung und Wartung erlauben wesentlich flexiblere Entwicklungsprozesse:
  - Dokumentation, Testen, Versionsverwaltung, Evolution, Generierung, Installation, ...

7. Implementierung
7.2. Codingstandards: Stilfragen der Codierung

Prof. Dr. Bernhard Rumpe
Software Systems Engineering
Technische Universität Braunschweig

http://www.sse-tubs.de/

Literatur:
- Vermeulen et al.:
  The Elements of Java Style
- Scott Ambler: http://www.ambysoft.com/
  javaCodingStandards.html
- Balzert Bd. 1 LE 33

Programmierstil

- Qualität von Programmcode:
  - Funktionalität
  - Verständlichkeit, Wartbarkeit
  - Effizienz
  - Eleganz
- Im Ablauf identische Programme können in der Verständlichkeit erheblich differieren.
  - Programmierkonventionen, z.B.:
    - Verwendete Konstrukte
    - Reihenfolgen
    - Klammerung
  - Bezeichnerwahl
  - Layout
    - Einrückung
- "Style Guide": Standard-Konventionen (z.B. für Projekt, Firma)

Was tut dieses Java-Programm?

```
public class Z
{public static void main(String[] args)
{double x = Console.readDouble("X:"); double z = Console.readDouble("Z:") /
   100; int l = Console.readInt("L:");
double y; for (y = z - 0.01; y <= z + 0.01; y += 0.00125)
{double p = x * y/12/(1 - (Math.pow(1/(1 + y/12),l*12)));
System.out.println(100*y+" : "+p);
}}}
```

Künstlich verschlechtertes (!) Beispiel aus:
Horstmann/Cornell, Core Java Vol. I, Prentice-Hall 1997

Formatierungs-Richtlinien

```
public class Z {

  public static void main(String[] args) {
    double x;
    double z;
    int l;
    x = Console.readDouble("X:");
    z = Console.readDouble("Z:") / 100;
    l = Console.readInt("L:");
    double y;
    for (y = z - 0.01; y <= z + 0.01; y += 0.00125){
      double p = x * y /12 /
        (1 - (Math.pow(1/(1 + y / 12), l * 12)));
      System.out.println(100*y+" : "+p);
    }
  }

}
```

Hinweise zur Formatierung (a)

- Einheitliche Formatierung verwenden !
  - Werkzeuge ("pretty printer", "beautifier")

- Gemäß Schachtelungstiefe einrücken
  - Genau festgelegte Anzahl von Leerzeichen
    (keine Tabulatoren! Definitiv keine Tabulatoren!)

  - Formatierungsprobleme bei zu tiefer Schachtelung deuten oft auf Strukturprobleme des Codes hin !

- Leerzeilen verwenden (einheitlich)
  - z.B. vor und nach Methoden
  - Aber: Zusammenhängender Code soll auf einem normalen Bildschirm darstellbar bleiben!

Hinweise zur Formatierung (b)

- Leerzeichen verwenden (einheitlich):
  - z.B. Operatoren und Operanden durch ein Leerzeichen trennen
  - z.B. keine Leerzeichen vor Methodenparametern, nach Casts

- Einheitliche Dateistruktur verwenden
  - z.B.: Je Klasse eine .java-Datei, je Package ein Verzeichnis
  - "package"-Statement immer als erste Zeile (noch vor "import")
  - Zuerst die Methoden, dann die Attribute
    (oder umgekehrt, aber einheitlich!)

Einrückung

- JavaSoft-Konvention zu Einrückungstiefe:
  - 4 Zeichen normale Einrückung
  - 8 Zeichen Einrückung zu Sonderzwecken

- Wichtig:
  - **Schreiben aus der Sicht des Lesers!**
  - denn Code wird wesentlich(!) öfter gelesen als geschrieben.

  - Und: Leseaufwand bei schlechtem Code ist immens.

Beispiele zur Einrückung

- JavaSoft-Konvention für lange Methodenköpfe:

```
void methode (int x, Object y, String z, Xyz v,
              float p);  //konventionell
```

```
private static synchronized void etwasLang
        (int x; Object y, String z, Xyz v,
        float p) {
        // Acht Zeichen Einrückung hier besser
    x = ... // Methodenrumpf nur vier eingerückt
```

Beispiele zur Einrückung

- JavaSoft-Konvention für die Fallunterscheidung:

```
// nicht gut erfassbar
if ((bedingung1 && bedingung2 && bedingung3)
    || bedingung4) {
    codeFürKomplexeBedingung(); …
```

```
// besser
if ((bedingung1 && bedingung2 && bedingung3)
        || bedingung4) {
    codeFürKomplexeBedingung(); …
```

Beispiele zu Klammern und Separatoren

- Relativ schlecht wartbarer Code (Java):

```
if (bedingung)
    methode();
```

- Besser wartbarer Code:

```
if (bedingung) {
    methode();
}
```

- Relativ schlecht wartbarer Code (Pascal):

```
if xyz then
    begin
      statement1;
      statement2
    end;
```

- Besser wartbarer Code:
  - Strichpunkt nach `statement2`

Wahl von Bezeichnern

- Einheitliche Namenskonvention verwenden!
- Bezeichner sollen:
  - natürlicher Sprache entnommen sein (bevorzugt Englisch)
    - Ausnahmen: Schleifenvariablen, manche Größen in Formeln
  - aussagekräftig sein
  - leicht zu merken sein
  - nicht zu lang sein, wenn häufig verwendet
  - Kurze Bezeichner nur bei sehr kleinem Scope (Schleifenvariablen)
- Beispiele: wofür ist welcher Bezeichner gut?
  - x1, x2, i, j, k
  - customername, CustomerName, customerName, cust_name
  - kontoOeffnen, oeffneKonto, kontoOeffnung, kontoGeoeffnet
  - CONSTANT

⇨ discuss

Beispiele für Namenskonventionen (a)

- Klasse:
  - Substantiv, erster Buchstabe groß, Rest klein
  - Ganze Worte, Zusammensetzung durch Großschreibung
  - Bsp: `Account, StandardTemplate`

- Methode:
  - Verb, Imperativ (Aufforderung), erster Buchstabe klein
  - Lesen und Schreiben von Attributen mit get/set-Präfix im Namen
  - Bsp:
    `checkAvailability(), doMaintenance(), getDate()`
  - Abfragen/Queries (bool-Ergebnis, keine Änderungen):
    `isLarge(), hasFather()`

Beispiele für Namenskonventionen (b)

- Konstanten:
  - Nur Großbuchstaben, Worte mit "_" zusammengesetzt
  - Standardpräfixe: "MIN_", "MAX_", "DEFAULT_", …
  - Bsp.: `NORTH, BLUE, MIN_WIDTH, MAX_WIDTH, DEFAULT_SIZE,`

- Attribute
  - Mit führendem Underscore
  - Bsp: `_availability, _date`

Lesbarkeit durch Bezeichnerwahl

```
public class Zinstabelle {

  public static void main(String[] args) {
    double betrag;       //zu verzinsender Betrag
    double zinssatzJahr; //jaehrlicher Zins als Faktor
    int laufzeit;        //Laufzeit in Jahren

    betrag = Console.readDouble("Betrag:");
    zinssatzJahr = Console.readDouble("Zinssatz:") / 100;
    laufzeit = Console.readInt("Laufzeit:");

    double y;

    for (y = zinssatzJahr - 0.01;
        y <= zinssatzJahr + 0.01; y += 0.00125){
      double zinssatzMonat = y/12;
      double zahlung = betrag * zinssatzMonat /
          (1 - (Math.pow(1/(1 + zinssatzMonat),
          laufzeit * 12)));
      System.out.println(100*y+" : "+zahlung);
    }
  }

}
```

Änderungsfreundlicher Code (1)

- Wahl von Variablen, Konstanten und Typen orientiert an der fachlichen Aufgabe, nicht an der Implementierung:
  - Gutes Beispiel (C):

```
typedef char name [nameLength]
typedef char firstName [firstNameLength]
```

  - Schlechtes Beispiel (C):

```
typedef char string10 [10]
```

- Symbolische Konstante statt literale Werte verwenden, wenn spätere Änderung denkbar (also immer).
- Algorithmen, Formeln, Standardkonzepte in Methoden/Prozeduren kapseln.
- An den Leser denken:
  - Zusammenhängende Einheit möglichst etwa Größe eines typischen Editorfensters (40-60 Zeilen, 70 Zeichen breit)
  - Text probehalber vorlesen ("Telefon-Test")

Änderungsfreundlicher Code (2)

- Strukturierte Programmierung
  - Kein "goto" verwenden (soweit noch vorhanden)
  - "switch" nur mit "break"-Anweisung nach jedem Fall
  - "break" nur innnerhalb "switch"-Anweisungen verwenden
  - "continue" nicht verwenden
  - "return" nur zur Rückgabe des Werts, nicht als Rücksprung in der Mitte der Methode

- „Goto considered harmful“ (E.W. Dijkstra)
  - beschreibt die Probleme der Spagettiprogrammierung

Änderungsfreundlicher Code (2)

- Übersichtliche Ausdrücke:
  - Möglichst seiteneffektfreie Ausdrücke
    - Schlechtes Bsp.: `y += 12*x++;`
    - Inkrementierung/Dekrementierung besser in separaten Anweisungen
      - ++x; y += 12*x
  - Fallunterscheidungsoperator (ternäres "? : ") sparsam einsetzen

- Sichtbarkeitsprüfungen des Compilers ausnutzen:
  - Attribute möglichst lokal und immer "private" deklarieren
  - Wiederverwendung "äußerer" Namen (Verschattung) vermeiden

Stilistisch verbessertes Programm

```
public class Zinstabelle {

  private static double zinsFormel
        (double betrag, double zinssatzMonat, double laufzeit) {
    double faktor = 1-Math.pow(1/(1+zinssatzMonat), 12*laufzeit)
    return betrag * zinssatzMonat / faktor;
  }

  static final double BEREICH = 0.01;
  static final double SCHRITT = 0.00125;

  public static void main(String[] args) {

    double betrag;       // zu verzinsender Betrag
    double zinssatzJahr; // jaehrlicher Zins als Faktor
    int laufzeit;        // Laufzeit in Jahren

    // Werte einlesen
    betrag = Console.readDouble("Betrag:");
    zinssatzJahr = Console.readDouble("Zinssatz:") / 100;
    laufzeit = Console.readInt("Laufzeit:");

    // Ergebnisse ausgeben
    double y;
    for (y = zinssatzJahr - BEREICH;
        y <= zinssatzJahr + BEREICH; y += SCHRITT){
      System.out.println(100*y+" : "+zinsFormel(betrag,y/12,laufzeit));
    }
  }
}
```

(Übrigens: es ist noch einiges verbesserungsfähig!)

Kommentare

- Prinzip der integrierten Dokumentation:
  - Kommentare im Code sind leichter zu warten
  - Kommentare sollten parallel zum Code entstehen
    - "Nach-Dokumentation" funktioniert in der Praxis nie!
  - Werkzeuge zur Generierung von Dokumentation (z.B. javadoc)
- Idealzustand:
  - Kommentare zu Klassen und Methoden stellen eine kompakte Spezifikation des Codes dar
- Kommentare sollen *nicht:*
  - den Code unlesbar machen
    - z.B. durch Verzerrung des Layouts
  - redundante Information zum Code enthalten
  - Schlechter Kommentar:

```
i++; // i wird hochgezaehlt
```

- Lesbarer kommentarfreier Code ist besser als formal kommentierter, aber unlesbarer Code.

Typischer Einsatz von Kommentaren

- "Vorspann" von Paketen, Klassen, Methoden etc.
  - Zweck, Parameter, Ergebnisse, *Exceptions*
  - Vorbedingungen, Abhängigkeiten (z.B. Plattform), Seiteneffekte
  - Version, Änderungsgeschichte, Status
- Formale Annahmen *(assertions):*
  - Vorbedingungen, Nachbedingungen
  - Allgemeingültige Annahmen (Invarianten)
- Lese-Erleichterung
  - Zusammenfassung komplexer Codepassagen
  - Überschriften zur Codegliederung
- Erklärung von einzelnen Besonderheiten des Codes
  - z.B. schwer verständliche Schritte, Seiteneffekte
- Arbeitsnotizen
  - Einschränkungen, bekannte Probleme
  - Offene Stellen (“!!!“,“TODO“), Anregungen, Platzhalter

Hinweise zum Verfassen von Kommentaren

- Phrasen statt Sätze: kein Problem!
  - Kürze und Übersicht zählt hier mehr als literarischer Anspruch.
- Deskriptiv (3. Person), nicht preskriptiv (2. Person)
  - Bsp: "Setzt die Kontonummer." statt: "Setze die Kontonummer."
- Unnötigen Rahmentext vermeiden:
  - Bsp.: "Setzt die Kontonummer." statt:
    "Diese Methode setzt die Kontonummer"
- Verwendung von "this" bzw. "diese/r/s"
  - Bsp: "Ermittelt die Version dieser Komponente." statt:
    "Ermittelt die Version der Komponente."

Professionell kommentierter Code

```
/*
 * @(#)Observer.java      1.14 98/06/29
 * Copyright 1994-1998 by Sun Microsystems, Inc., ...
 */
package java.util;

/**
 * A class can implement the <code>Observer</code> interface when it
 * wants to be informed of changes in observable objects.
 *
 * @author  Chris Warth
 * @version 1.14, 06/29/98
 * @see     java.util.Observable
 * @since   JDK1.0
 */
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param   o     the observable object.
     * @param   arg   an argument passed to the
     *                <code>notifyObservers</code> method.
     */
    void update(Observable o, Object arg);
}
```

Elements of Programming Style

Auszüge aus: Kernighan/Plauger,
The Elements of Programming Style, McGraw-Hill 1978 (!)

- Write clearly - don't be too clever.
- Don't sacrifice clarity for efficiency.
- Each module should do one thing well.
- Make sure every module hides something.
- Use the good features of a language, avoid the bad ones.
- Use the "telephone test" for readability.
- Don't stop with your first draft.
- Don't patch bad code - rewrite it.
- Don't comment bad code - rewrite it.
- Make it right before you make it faster.
- Make it clear before you make it faster.
- Keep it right when you make it faster.
- Let your compiler do the simple optimizations.
- Measure your program before making "efficiency" changes.

Viele dieser Stilelemente wurden z.B. in Extreme Programming adoptiert!

Elements of Java Style (a)

- Auszüge aus: Vermeulen et al.,
  The Elements of Java Style, Cambridge University Press 2000
- Bezeichnerwahl:
  - Join the vowel generation. ("appendSignature" statt "appdSgtr")
  - Capitalize only the first letter in acronyms. ("loadXmlDocument")
  - Pluralize the names of classes that group static services or constants.
  - Follow the JavaBeans conventions for property access ("get"/"set").

Elements of Java Style (b)

- Programmierung:
  - Follow the "Open/Closed" principle: Software entities should be open for extension but closed for modification.
  - Use assertions to test pre- and postconditions of a method.
  - Exceptions:
    - Unchecked (runtime) exceptions for serious unexpected errors.
    - Checked exceptions for errors that may appear in normal program execution.

Zusammenfassung
7.2. Stilfragen der Codierung

- Es gibt (leider) keinen allgemein anerkannten einheitlichen Codingstandard.
- Wichtig ist daher zunächst die Projekt-/Firmen-interne Einigung auf einen Standard!
- Wesentliche Kriterien:
  - Aussagekräftige und kompakte Kommentierung,
  - Namenswahl,
  - Einrückung
  - Größe von Codeblöcken (Methoden)
  - Größe von Modulen (Klassen)
- Java bietet im Internet zugängliche Standards (die weitgehend kompatibel sind)
- Codingstandards verhindern auch „Hacker-Code“

7. Implementierung
7.3. Datenstrukturen

Prof. Dr. Bernhard Rumpe
Software Systems Engineering
Technische Universität Braunschweig

http://www.sse-tubs.de/

Literatur:
- einschlägige Java-Bücher, z.B.
  David Flanagan: Java in a Nutshell

Bedeutung von Datenstrukturen

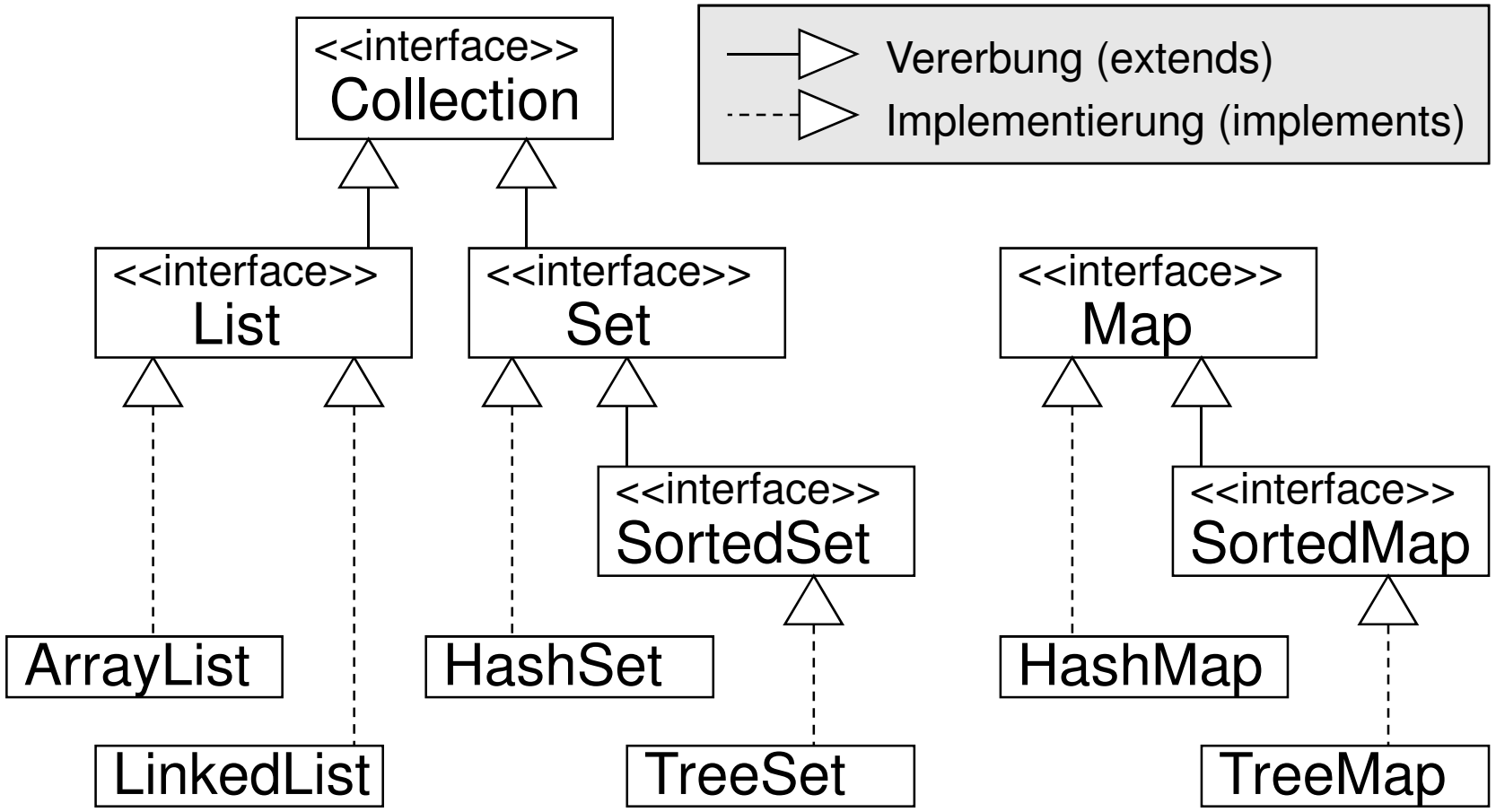
- Struktur
  - Ordnungssystem für die Daten
  - Bereitstellung von Standard-Funktionalität
- Wiederverwendung
  - Klassenbibliotheken
  - Standardalgorithmen
- Anpassbarkeit
  - Alternative Implementierungen für gleiche abstrakte Schnittstelle
- Optimierung
  - Alternativen mit verschiedener Leistungscharakteristik

- Beispiel: Implementierungen der Sets in Java

Abstrakter und konkreter Datentyp

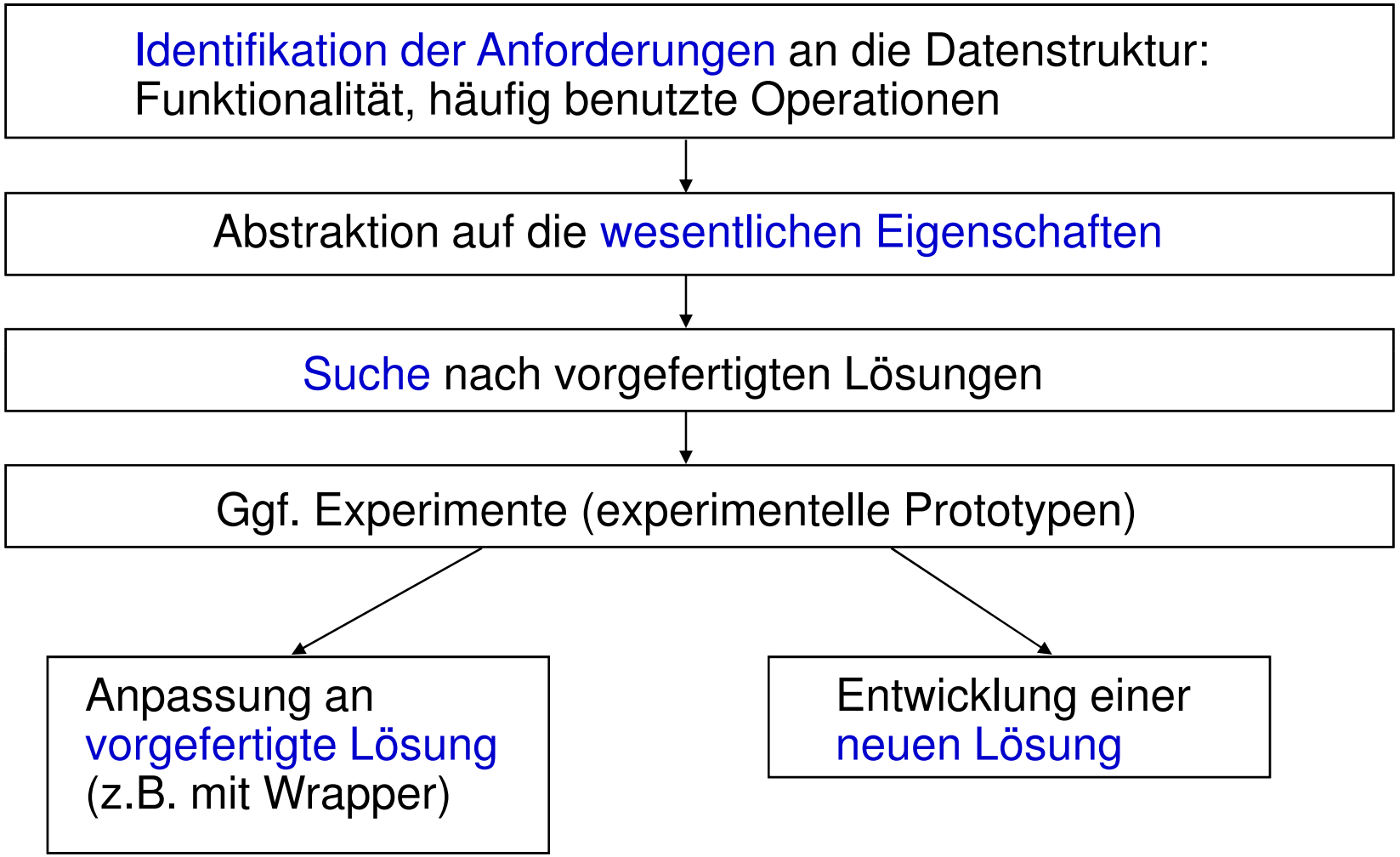
Abstrakter Datentyp (Schnittstelle)	***Konkreter Datentyp (Implementierung)***
Abstraktion: • Operationen • Verhalten der Operationen *Theorie:* • Algebraische Spezifikationen • Axiomensysteme *Praxis:* • Abstrakte Klassen • Interfaces *Beispiel:* • List	Konkretisierung: • Instantiierbare Klassen • Ausführbare Operationen ▪ *Theorie:* • Datenstrukturen • Effizienzfragen ▪ *Praxis:* • Alternative Implementierungen ▪ *Beispiel:* • Verkettete Liste • Liste durch Array

Auszug aus dem Collection-Framework



- Daneben gibt es WeakHashMap,
  und die veralteten Vector für List und Hashtable für Map

Vorgehensweise beim Datenstruktur-Entwurf



Suche nach vorgefertigten Lösungen