TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 14
Fehlersuche und fehlervermeidender Entwurf

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Fehlersuche

Die bisherigen Betrachtungen zur Korrektheit von Programmen bilden die Basis für jedwede systematische Fehlersuche – egal, ob man sich das bei der Fehlersuche bewusst macht oder nicht. Man sollte es sich aber besser bewusst machen; dafür ist dieser Abschnitt gedacht.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

Von einer eher abstrakten Ebene her lässt sich das Ganze eigentlich recht einfach durchschauen, und genau das machen wir hier vorrangig.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Auftreten*** **des Fehlers vs.** ***Ursache*** **des Fehlers.**
  - ➢ **Fehlerursache zurückverfolgen.**
- **Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.**
- **Wie finden: Assert-Anweisungen.**
- **Wenn Fehlerursache nicht klar:**
  - ➢ **Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.**
  - ➢ **Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.**

Zunächst machen wir uns klar, dass die Stelle, an der ein Fehler tatsächlich auftritt, nicht unbedingt die Stelle ist, an der der Fehler passiert ist. Das sichtbare Auftreten ist häufig nur ein *Symptom* der eigentlichen Ursache.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

Es liegt in der Natur von kausalen Zusammenhängen, dass die Ursache zeitlich *vor* dem Symptom passieren muss. Daher muss die Fehlerursache, ausgehend vom Auftreten, rückwärts in der Zeit gesucht werden.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

Mit unserem ganzen Begriffsapparat von Vor- und Nachbedingungen, Invarianten und Varianten lässt sich genau sagen, wo die Fehlerursache zu finden ist: an dem Zeitpunkt im Ablauf des Programms, an dem zum ersten Mal eine solche Zusicherung nicht erfüllt ist. Dabei ist es egal, ob das eine Darstellungs- oder Implementationsinvariante, eine Vor- oder Nachbedingung einer Subroutine oder eine Schleifeninvariante oder Schleifenvariante ist.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

***Erinnerung*: In Kapitel 05 hatten wir einen Abschnitt zu Throwable, Error und Assert-Anweisung. Wir hatten gesehen: Assert-Anweisungen sind Tests, die beim Kompilieren an- oder abgeschaltet werden können, so dass sie also bei der Fehlersuche verwendet und im produktiven Einsatz übersprungen werden können, ohne dass man sie mühsam einzeln aus dem Quelltext entfernen oder einzeln im Quelltext auskommentieren muss.**

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

Nun wird man nicht jede Zusicherung von vornherein durch eine Assert-Anweisung absichern wollen, das wäre in der Regel ein riesiger Haufen Arbeit, der sich wahrscheinlich nie auszahlen wird. Oft wäre das nicht einmal möglich; denken Sie etwa an die Vorbedingung von Subroutine find-zero, dass die als Parameter übergebene reellwertige Funktion stetig ist. Wie wollen Sie diese Vorbedingung in eine Assert-Anweisung umsetzen?

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

Aus diesen Gründen werden Sie in der Regel bei der Suche nach einem Fehler weitere Assert-Anweisungen einfügen müssen, mit denen Zusicherungen überprüft werden, die Sie bislang noch nicht überprüft hatten. Wenn Sie sich schon die Arbeit gemacht haben, diese weiteren Assert-Anweisungen zu schreiben, ist es sicherlich auch sinnvoll, sie auf Dauer im Quelltext zu belassen, daher steht in diesem Unterpunkt noch nicht das Wort „temporär“.

Nicht selten wird dabei auffallen, dass sie noch eine wichtige Zusicherung allein deshalb bisher nicht mit einer Assert-Anweisung überprüft haben, weil Sie sie bislang schlicht nicht auf dem Radar hatten, also weil Ihre Vorstellung von den Zusicherungen, die die Korrektheit Ihres Programms garantieren, lückenhaft ist.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Auftreten* des Fehlers vs. *Ursache* des Fehlers.
  - ➢Fehlerursache zurückverfolgen.
- Fehlerursache: wo zum ersten Mal irgendeine Vor- oder Nachbedingung, Invariante oder Variante nicht erfüllt ist.
- Wie finden: Assert-Anweisungen.
- Wenn Fehlerursache nicht klar:
  - ➢Weitere Assert-Anweisungen, die schrittweise noch weiter zurückgehen.
  - ➢Temporäre Assert-Anweisungen, die für das Datenbeispiel funktionieren (sollten!), mit dem der Fehler verfolgt wird.

Sie haben ja sicherlich mindestens ein konkretes Datenbeispiel in der Hand, bei dem der Fehlerfall aufgetreten ist. Falls nicht, sollten Sie systematisch versuchen, ein solches Datenbeispiel zu finden, und dann auch speichern, um damit weiterzuarbeiten.

Solange Sie nur mit einem einzelnen Datenbeispiel zugange sind, um einen Fehler aufzuspüren, können Sie auch mit Assert-Anweisungen arbeiten, die nicht allgemein, sondern nur für dieses Datenbeispiel erfüllt sein sollten. Ein einfaches Beispiel ist, dass Sie explizit mit einer Assert-Anweisung überprüfen, ob die Rückgabe einer Subroutine genau das ist, was die Subroutine für dieses Datenbeispiel zurückliefern sollte.

Solche Assert-Anweisungen sollten natürlich nur temporär im Quelltext sein, bis Sie den Fehler gefunden haben. Um sie danach aus dem Quelltext zu entfernen, überlegen Sie sich eine unverwechselbare Zeichenkette Ihrer Wahl und schreiben Sie diese jeweils in einem Kommentar zu jeder Assert-Anweisung hinzu, die Sie nicht permanent im Quelltext belassen, sondern nach der Fehlersuche wieder entfernen wollen. Durch Textsuche nach dieser Zeichenkette finden Sie alle diese Assert-Anweisungen schnell wieder.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Als erstes Fallbeispiel noch einmal die Funktion find-zero in Racket beziehungsweise Methode findZero in Java.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Eine derart grundlegende Funktion wird in der Regel vielfach in verschiedensten Anwendungsfällen verwendet und damit automatisch auch in vielfältiger Weise getestet.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Irgendwo tritt jetzt der Fehler auf. dass ein berechneter Fixpunkt gar kein Fixpunkt ist.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Ein Fixpunkt einer Funktion *f* ist bekanntlich ein Wert *x* aus dem Definitionsbereich von *f*, so dass *f*(*x*) gleich *x* ist. Bei reellen Zahlen müssen wir uns wieder damit begnügen, dass *f*(*x*) nur *ungefähr* gleich *x* ist, das heißt, um sicher zu sein, dass *x* kein Fixpunkt ist, muss *f*(*x*) entsprechend weit weg von *x* sein.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Man berechnet häufig einen Fixpunkt einer Funktion *f*, indem man eine Nullstelle derjenigen Funktion berechnet, die aus *f* gebildet wird, indem *x* von *f*(*x*) abgezogen wird.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Der naheliegende Gedanke ist natürlich, dass die Funktion find-zero fehlerhaft ist, da sie ja ein fehlerhaftes Ergebnis zurückliefert.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Das kann man natürlich überprüfen, indem man find-zero mit möglichst repräsentativen Funktionen austestet. Idealerweise hat man diese Tests längst schon als JUnit-Test realisiert und kann sie einfach laufen lassen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Der Verdacht keimt auf, dass vielleicht irgendetwas mit dieser speziellen Funktion *f* falsch ist. Also schaut man sich die Werte der Funktion *f* rund um die angebliche Nullstelle an und Bingo: Diese Stelle ist eine Sprungstelle von *f*, *f* ist also gar nicht wie verlangt stetig.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Erstes Fallbeispiel: Methode find-zero

- **Wird als untergeordnete Methode unzählige Male in diversen großen numerischen Programmpaketen verwendet.**
- **Irgendwann irgendwo fällt auf, dass irgendein Wert *x* nicht wie gedacht ein Fixpunkt einer Funktion *f* ist.**
  - **Mit anderen Worten: *f*(*x*) ist viel zu weit weg von *x*.**
- **Zurückverfolgung: find-zero hat für die Funktion *x* → *f*(*x*) – *x* einen Wert berechnet, der keine Nullstelle ist.**
  - **Ist also find-zero fehlerhaft?**
  - **Systematische Tests mit verschiedenen stetigen Funktionen sagen nein.**
- **Zusätzliche Assert-Anweisung zum Test liefert die Erkenntnis: *x* → *f*(*x*) – x springt unstetig vom Positiven ins Negative.**
  - **Wird von find-zero als angebliche Nullstelle gefunden.**

Sie können leicht anhand des Quelltextes von find-zero verifizieren, dass im Falle eines Vorzeichenwechsels tatsächlich eine solche Unstetigkeitsstelle als vermeintliche Nullstelle von find-zero gefunden werden könnte.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Noch ein zweites Fallbeispiel, nun aus der Forschungsarbeit der Arbeitsgruppe des Autors dieser Folien. Ein solches Beispiel ist natürlich viel zu komplex, um es hier in allen Details durchzusprechen, daher nur kurz und in groben Zügen.

Es geht um das, was Sie am Fahrkartenautomaten oder auch auf dem entsprechenden Internetportal der Deutschen Bahn als erstes tun: sich potentiell attraktive Verbindungen ausgeben lassen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Das heißt, Sie geben erst einmal die Eckdaten ein: von wo nach wo Sie fahren wollen und wann ungefähr.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Das System weiß ja nicht, was für Bahnverbindungen für Sie persönlich attraktiv sind: wie viel Zeit und wie viel Unbequemlichkeit – etwa in Form von engen Umstiegen – Sie bereit sind in Kauf zu nehmen, um Geld zu sparen. Daher soll nicht nur ein einziger Verbindungsvorschlag ausgegeben werden, sondern mehrere, die möglichst alle Kundenprofile abdecken.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Wichtig ist, möglichst viele Optionen möglichst früh im Algorithmus auszusortieren, weil sonst die Antwortzeit schnell jenseits von gut und böse wäre. Das dafür verwendete Konzept ist in Informatik, Mathematik und Wirtschaftswissenschaften unter dem Namen *Pareto-Dominanz* oder kurz *Dominanz* bekannt.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Was heißt Dominanz nun konkret? Dazu vergleicht man zwei Verbindungen miteinander, die vom selbsn Startbahnhof zum selben Zielbahnhof fahren, also in der Gunst des Kunden potentiell in Konkurrenz zueinander stehen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Eine Verbindung, die früher abfährt, kann immer noch attraktiv sein, wenn sie auch früher ankommt. Umgekehrt kann eine Verbindung, die später ankommt, ebenfalls immer noch attraktiv sein, wenn sie später abfährt. Aber früher losfahren und später ankommen ist doch eher unattraktiv, ...

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

... es sei denn, es sprechen andere Kriterien dafür, *doch* diese Verbindung zu nehmen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

Ansonsten kann man wohl sicher davon ausgehen, dass Verbindung B in jedem denkbaren Kundenprofil attraktiver als Verbindung A ist, ...

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Gegeben***: **Start- und Zielbahnhof, Abfahrtstag, Zeitfenster für die Abfahrt.**
- ***Gesucht***: **möglichst attraktive Bahnverbindungen, die im Zeitfenster abfahren.**
- ***Mindestziel***: **keine Verbindung ausgeben, die durch eine andere Verbindung *dominiert* ist.**
- **Verbindung A ist *dominiert* von Verbindung B, wenn**
  - **A nicht später als B abfährt und nicht früher als B ankommt sowie**
  - **garantiert nicht billiger oder komfortabler ist.**
- **Eine dominierte Verbindung ist garantiert keine attraktive Verbindung!**
  - **Kann aus der Betrachtung entfernt werden.**

... so dass A dem Kunden sicher nicht angezeigt werden muss.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Wir werden nur sehr grob die algorithmische Vorgehensweise skizzieren.

***Vorgriff*: In der Lehrveranstaltung AuD im zweiten Fachsemester werden Sie den Algorithmus von Dijkstra kennen lernen. Der hier skizzierte Algorithmus zur Berechnung attraktiver Bahnverbindungen ist eine etwas kompliziertere Variante dieses eigentlich recht einfachen Algorithmus.**

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Der Algorithmus wird typischerweise als Schleife realisiert. Alternativ könnte er auch durch Rekursion realisiert sein, würde damit aber wohl wesentlich komplizierter werden.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Sie können sich den Algorithmus so vorstellen, dass eine ganze Reihe von Teilverbindungen vom Startbahnhof aus in alle Richtungen ausschwärmen und den Zielbahnhof suchen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Ausschwärmen heißt, dass die Teilverbindungen durch weitere Zugabschnitte immer weiter wachsen. Sie gabeln sich dabei auch auf, denn von einer Teilverbindung entstehen im Allgemeinen etliche weitere Teilverbindungen durch Hinzufügen verschiedener weiterer Zugabschnitte vom selben Bahnhof aus. In dem Zug, mit dem man zu einem Zwischenbahnhof gekommen ist, sitzenzubleiben, konstituiert *eine* mögliche Fortsetzung; jeder Zug, in den man stattdessen umsteigen könnte ist ebenfalls eine mögliche Fortsetzung.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Der entscheidende Punkt ist, dass dann am Zielbahnhof dieses Zugabschnitts nach weiteren Teilverbindungen geschaut wird, die bis zu diesem Bahnhof gekommen sind, und dass dann alle diese Teilverbindungen miteinander verglichen werden, um dominierte zu identifizieren und herauszunehmen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Die Schleife sollte so früh wie möglich abgebrochen werden, denn wenn Sie beispielsweise eine Verbindungsauskunft von Darmstadt nach Frankfurt haben wollen, dann soll die Schleife nicht erst beendet werden, wenn von Darmstadt aus das gesamte Zugnetzwerk bis Kapstadt und Wladiwostok durchsucht wurde.

Die Frage ist aber, wann die Schleife frühestens abgebrochen werden *darf*, also wann sichergestellt ist, dass keine potentiell attraktive Verbindung vom Startbahnhof zum Zielbahnhof unentdeckt geblieben ist.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Natürlich darf die Schleife nicht abbrechen, solange noch *gar keine* Verbindung vom Start- zum Zielbahnhof gefunden wurde.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- **Schleife:**
  - **verwaltet eine Menge von Teilverbindungen vom Startbahnhof zu anderen Bahnhöfen,**
  - **fügt in jedem Schleifendurchlauf einen Zugabschnitt an eine der Teilverbindungen an und**
  - **eliminiert die dominierten Teilverbindungen am neuen Endbahnhof dieser Teilverbindungen.**
- **Abbruch der Schleife:**
  - **mindestens eine Verbindung ist zum Zielbahnhof „durchgekommen“,**
  - **es wird garantiert keine weitere attraktive Verbindung mehr bis zum Zielbahnhof „durchkommen“.**

Irgendwann kommt aber der Punkt, an dem man sicher sagen kann, dass alle Teilverbingungen, die noch im Rennen sind, auf allen denkbaren Wegen nur noch zu *weniger* attraktiven Gesamtverbindungen bis zum Zielbahnhof vervollständigt werden können, weil sie schon als Teilverbindung zu viel Fahrzeit, Fahrpreis und Unbequemlichkeit kosten im Vergleich zu den bis dahin schon am Zielbahnhof angekommenen Gesamtverbindungen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler*: Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz*: Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers*: Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche*:**
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Kommen wir zum eigentlichen Thema dieses Abschnitts, der Fehlersuche.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Sie erinnern sich von der vorletzten Folie, wie Dominanz definiert war. Neben Fahrpreis und Bequemlichkeit war vor allem der Vergleich der Abfahrtzeiten und der Ankunftzeiten wichtig. Der Programmierfehler war: Der Vergleich der Ankunftzeiten wurde vergessen.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Damit war die Bedingung für Dominanz zu restriktiv. Das heißt, es wurden zu viele Teilverbindungen herausgenommen, darunter zuweilen auch solche, die bei korrekter Implementierung der Dominanzregel zu attraktiven Gesamtverbindungen verlängert worden wären.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Das Auftreten eines solchen Fehlers kann kaum systematisch geprüft werden. Wir reden ja von Tests mit zehntausenden Anfragen. Abgesehen von sehr kleinen, überschaubaren Fällen, in denen der Fehler kaum auftreten dürfte, weiß man ja in der Regel nicht, welche die attraktivsten Verbindungen für eine gegebene Anfrage sind, und kann diese auch nicht ´mal eben schnell per Augenschein selbst herausfinden. Daher kann man auch nicht so einfach sagen, ob die berechneten Verbindungen wirklich alle attraktiven umfassen oder ob welche fehlen. Und Fehler dieser Art produzieren letztendlich auch nur bei einer Minderzahl der unzähligen getesteten Anfragen ein fehlerhaftes Ergebnis, so dass man Fehler dieser Art nicht von vornherein ausschließen kann, indem man als Stichprobe ein paar Verbindungen zufällig auswählt und diese manuell auf Herz und Nieren prüft.

In vielen Anwendungen – so wie hier – ist es daher reiner Zufall, dass man überhaupt ein fehlerhaftes Ergebnis feststellt, zum Beispiel wenn man sich die Ergebnisse zu einer bestimmten Anfrage aus ganz anderen Gründen genauer anschaut und dann auf Ungereimtheiten stößt.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Aber auch nachdem festgestellt werden musste, dass die Ausgabe fehlerhaft war, gestaltet sich die sich daran anschließende Fehlersuche häufig alles andere als einfach.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Wir reden in der Praxis ja in der Regel von Programmen mit tausenden oder zehntausenden oder noch mehr Zeilen. Und wenn man keine Idee hat, worin der Fehler bestehen könnte, weiß man oft erst einmal gar nicht, wo man eigentlich suchen soll.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Was die Fehlersuche dann wirklich aufwendig macht, ist, dass die Datenmenge in jedem Durchlauf der Schleife gigantisch groß ist. Es war klar, dass irgendwo Teilverbindungen dominiert wurden, die nicht dominiert werden sollten. Aber welche der vielen Millionen waren das?

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweites Fallbeispiel: Bahnverkehrsauskunft

- ***Fehler***: **Versehentlich und unbemerkt wurde vergessen, die Ankunftzeiten beim Dominanztest miteinander zu vergleichen.**
- ***Konsequenz***: **Viele Teilverbindungen, die potentiell zu attraktiven Gesamtverbindungen hätten erweitert werden können, sind dominiert und daher aus dem Pool entfernt worden.**
- ***Auftreten des Fehlers***: **Zufällig fiel auf, dass attraktive Verbindungen in der Ausgabe fehlten.**
- ***Problem Fehlersuche***:
  - **sehr umfangreicher Quelltext,**
  - **Millionen von Teilverbindungen während der Schleife,**
  - **die effektivsten Assertion-Anweisungen wusste man dann hinterher...**

Gut, dass wir den Fehler überhaupt entdeckt haben, heißt ja, wir haben bei wenigstens einer Anfrage mindestens eine Gesamtverbindung entdeckt, die von keiner Gesamtverbindung in der Ausgabe dominiert wird, aber trotzdem nicht in der Ausgabe vorhanden ist. Man könnte also denken, dass man einfach nur die schrittweise Komposition dieser einzelnen Gesamtverbindung verfolgen müsste, um die Stelle im Programmablauf zu finden, an der der Fehler passiert ist.

Aber ganz so einfach ist das nicht, denn die Verbindung, deren Fehlen in der Ausgabe aufgefallen ist, muss ja noch nicht die sein, die wirklich in die Ausgabe gehört hätte. Es kann durchaus sein, dass diese Verbindung bei korrektem Programmlauf noch durch eine andere Verbindung dominiert worden wäre, die man überhaupt nicht auf dem Radar hat. Und so weiter.

Man stochert daher potentiell lange Zeit im Nebel herum, bis man endlich klar sieht.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- Fehler in Bibliothekskomponente?
- Falls undurchschaubar: verdächtige Quelltext-Passagen blind neu implementieren

Zum Abschluss dieses Abschnitts noch zwei allgemeine Gedanken.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- Fehler in Bibliothekskomponente?
- Falls undurchschaubar: verdächtige Quelltext-Passagen blind neu implementieren

Bei einer aufwändigen Fehlersuche werden Sie sicherlich immer wieder geneigt sein, den Fehler dann doch eher in dem Code zu vermuten, den Sie nicht selbst implementiert haben und den Sie auch nicht selbst systematisch austesten können, weil der Quelltext nicht verfügbar ist. Aller Erfahrung nach ist diese Vermutung höchstwahrscheinlich falsch, das heißt, wenn Sie sich noch weiter Mühe geben, werden Sie am Ende doch den Fehler bei sich selbst finden. Denn Bibliotheksroutinen sind in der Regel unzählige Male in verschiedensten Kontexten verwendet worden, so dass eigentlich alle Fehler längst gefunden und eliminiert sein müssten.

Nichtsdestotrotz gibt es natürlich eine Restwahrscheinlichkeit dafür, dass eine Bibliotheksroutine fehlerhaft ist. Zum Beispiel könnte es ein kürzliches Update gegeben haben, das eben noch *nicht* unzählige Male verwendet wurde. Nicht selten ist auch ein Fehler schon längere Zeit bekannt, bevor er behoben wird.

Fehlersuche

TECHNISCHE UNIVERSITÄT DARMSTADT

- Fehler in Bibliothekskomponente?
- Falls undurchschaubar: verdächtige Quelltext-Passagen blind neu implementieren

Zuweilen kommen Sie nicht mit vertretbarem Zeitaufwand an den Fehler heran. In einem solchen Fall kann es ökonomisch vorteilhaft sein, nicht mehr weiter nach dem Fehler zu suchen, sondern die Passagen, in denen der Fehler sein könnte, möglichst unbelastet noch einmal neu zu programmieren, idealerweise ist das nicht einmal derselbe Programmierer.

Ob das in einer konkreten Situation ökonomisch vorteilhaft ist oder nicht, ist eine Ermessensentscheidung, für die man wohl keine allgemeinen Grundsätze sinnvoll aufstellen kann.

TECHNISCHE UNIVERSITÄT DARMSTADT

Laufzeittests

Fehler sollen natürlich nach Möglichkeit nicht erst im realen Einsatz auftreten. Daher führt man in der Entwicklungsphase umfangreiche Korrektheitstests durch. Die konzeptuelle Basis haben wir schon entwickelt mit den verschiedenen Arten von Zusicherungen.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Black Box: JUnit-Tests (und Assert-Anweisungen)
  - Darstellungsinvarianten von Klassen und Interfaces
  - Nachbedingungen von Subroutinen
- White Box: Assert-Anweisungen
  - Implementationsinvarianten von Klassen
  - Vor- und Nachbedingungen von einzelnen Anweisungen
  - Schleifen(in)varianten

Man unterscheidet generell zwischen zwei Arten von Tests: Black Box und White Box.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Black Box: JUnit-Tests (und Assert-Anweisungen)
  - Darstellungsinvarianten von Klassen und Interfaces
  - Nachbedingungen von Subroutinen
- White Box: Assert-Anweisungen
  - Implementationsinvarianten von Klassen
  - Vor- und Nachbedingungen von einzelnen Anweisungen
  - Schleifen(in)varianten

Black Box heißt, dass man eine Einheit als Ganzes von außen testet, ohne in ihren Code hineinzuschauen. Mit unserem Begriffsapparat bedeutet das Folgendes:

Bei Referenztypen wird geprüft, ob nach jedem Methodenaufruf die Darstellungsinvariante wieder erfüllt ist; dafür müssen nur die public-Methoden aufgerufen werden, wenn diese ausreichend umfangreich Zugriff gewähren. Beispielsweise bei einer Klasse, die Interface List aus java.util implementiert, kann man die Darstellungsinvariante jederzeit einfach dadurch abprüfen, dass man den momentanen Inhalt der Liste mit dem vergleicht, was die Liste in diesem Moment eigentlich enthalten sollte, wenn alles korrekt ist.

Und bei Subroutinen wird entsprechend die Nachbedingung geprüft durch Aufrufe, in denen die Vorbedingung erfüllt ist.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Black Box: JUnit-Tests (und Assert-Anweisungen)
  - Darstellungsinvarianten von Klassen und Interfaces
  - Nachbedingungen von Subroutinen
- White Box: Assert-Anweisungen
  - Implementationsinvarianten von Klassen
  - Vor- und Nachbedingungen von einzelnen Anweisungen
  - Schleifen(in)varianten

Wie das Wort Unit in JUnit-Test schon nahelegt, sind solche Unit-Tests nur für Black-Box-Testing die Methode der Wahl.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Black Box: JUnit-Tests (und Assert-Anweisungen)
  - Darstellungsinvarianten von Klassen und Interfaces
  - Nachbedingungen von Subroutinen
- White Box: Assert-Anweisungen
  - Implementationsinvarianten von Klassen
  - Vor- und Nachbedingungen von einzelnen Anweisungen
  - Schleifen(in)varianten

Es kann aber auch nicht schaden, an den Stellen im Quelltext, an denen ein Referenztyp oder eine Subroutine verwendet wird, eine Assert-Anweisung einzufügen, die nach Verwendung dann die Darstellungsinvariante der Klasse beziehungsweise die Nachbedingung der Subroutine testet.

JUnit-Tests haben den Vorteil, dass man die Eingaben für die zu testende Einheit systematisch variieren kann, um möglichst alle denkbaren Fälle abzudecken. Assert-Anweisungen im Quelltext haben hingegen den Vorteil, dass sie im – echten oder simulierten – praktischen Einsatz verwendet werden können und dann logischerweise genau in den Situationen zum Einsatz kommen, die in der konkreten Anwendung vorkommen.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Black Box: JUnit-Tests (und Assert-Anweisungen)
  - Darstellungsinvarianten von Klassen und Interfaces
  - Nachbedingungen von Subroutinen
- White Box: Assert-Anweisungen
  - Implementationsinvarianten von Klassen
  - Vor- und Nachbedingungen von einzelnen Anweisungen
  - Schleifen(in)varianten

Die anderen Zusicherungen, die wir besprochen hatten, beziehen sich auf die Interna von Referenztypen beziehungsweise Subroutinen und gehören daher zum Thema White-Box-Testing.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Testumgebung parallel zum eigentlichen Code entwickeln:
  - check-expect usw. in Racket,
  - JUnit-Tests und Assert-Anweisungen in Java.
- Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung ***unverzüglich*** mit ändern / weiterentwickeln.
  - Der Code geht ***immer*** fehlerfrei durch die Tests durch.
- Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.

Nach dieser Begriffsklärung nun die allgemeine Vorgehensweise. Die anerkannt einzig richtige Vorgehensweise ist einfach zu formulieren, erfordert aber ein hohes Maß an Disziplin.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Testumgebung parallel zum eigentlichen Code entwickeln:**
  - **check-expect usw. in Racket,**
  - **JUnit-Tests und Assert-Anweisungen in Java.**
- **Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung *unverzüglich* mit ändern / weiterentwickeln.**
  - **Der Code geht *immer* fehlerfrei durch die Tests durch.**
- **Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.**

Man ist vielleicht geneigt, zuerst einmal drauflos zu programmieren und dann hinterher zu testen. Aller Erfahrung nach empfehlenswert ist aber eher, Software und Testumgebung konsequent parallel zu entwickeln, so dass beide immer auf demselben Stand sind und zusammenpassen. So kann das Testen nicht aus Zeitgründen am Ende unter den Tisch fallen.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Testumgebung parallel zum eigentlichen Code entwickeln:
  - ➢ check-expect usw. in Racket,
  - ➢ JUnit-Tests und Assert-Anweisungen in Java.
- Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung ***unverzüglich*** mit ändern / weiterentwickeln.
  - ➢ Der Code geht ***immer*** fehlerfrei durch die Tests durch.
- Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.

Sie erinnern sich: Schon bei der allerersten selbstdefinierten Funktion und dann auch bei allen weiteren in Racket hatten wir Wert auf check-expect und die anderen Check-Funktionen gelegt.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Testumgebung parallel zum eigentlichen Code entwickeln:
  - ➢ check-expect usw. in Racket,
  - ➢ JUnit-Tests und Assert-Anweisungen in Java.
- Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung ***unverzüglich*** mit ändern / weiterentwickeln.
  - ➢ Der Code geht ***immer*** fehlerfrei durch die Tests durch.
- Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.

Es wäre sicherlich auch bei Java sinnvoll gewesen, Fehlerbehandlung in der FOP von Anfang an immer konsequent zu integrieren. Aber aufgrund der Komplexität der Thematik in Java haben wir hier einen Kompromiss geschlossen und Fehlerbehandlung erst im Kapitel 05 behandelt.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Testumgebung parallel zum eigentlichen Code entwickeln:
  - ➢ check-expect usw. in Racket,
  - ➢ JUnit-Tests und Assert-Anweisungen in Java.
- Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung ***unverzüglich*** mit ändern / weiterentwickeln.
  - ➢ Der Code geht ***immer*** fehlerfrei durch die Tests durch.
- Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.

Die parallele Entwicklung von Code und Testumgebung bedeutet natürlich, die Testumgebung jedes Mal anzupassen, wenn man den eigentlichen Code so ändert, dass die bisherige Testumgebung nicht mehr passt.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Testumgebung parallel zum eigentlichen Code entwickeln:
  - ➢ check-expect usw. in Racket,
  - ➢ JUnit-Tests und Assert-Anweisungen in Java.
- Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung ***unverzüglich*** mit ändern / weiterentwickeln.
  - ➢ Der Code geht ***immer*** fehlerfrei durch die Tests durch.
- Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.

Das muss das Ziel sein: immer auf der sicheren Seite sein, dass alles korrekt ist – zumindest soweit es von der Testumgebung geprüft wird.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Testumgebung parallel zum eigentlichen Code entwickeln:
  - ➢ check-expect usw. in Racket,
  - ➢ JUnit-Tests und Assert-Anweisungen in Java.
- Bei jeder Änderung / Weiterentwicklung des eigentlichen Codes auch die Testumgebung ***unverzüglich*** mit ändern / weiterentwickeln.
  - ➢ Der Code geht ***immer*** fehlerfrei durch die Tests durch.
- Für jeden potentiell bedeutsamen gefundenen Fehler einen neuen Test einbauen, der diesen findet.

Wann immer man einen Fehler findet und behebt, bietet es sich an, das Nichtvorhandensein dieses Fehlers von nun an durch einen weiteren Test zu überprüfen. Denn es kann schon einmal passieren, dass man einen Fehler, den man einmal gefunden und behoben hat, durch spätere Weiterentwicklungen des Codes versehentlich wieder einbaut.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Coverage:

- **Randfälle**
- **Bei Verzweigungen: jeder mögliche Pfad**

Das wichtigste Kriterium für Laufzeittests ist, inwieweit alle denkbaren Pfade des Prozesses durch das Programm abgedeckt werden. Natürlich ist es unmöglich, wirklich jeden denkbaren Ablauf des Programms testweise zu durchlaufen. Aber anzustreben ist, dass eine Menge von möglichen Programmläufen durchgetestet wird, so dass man mit großer Sicherheit sagen kann: Wenn alle diese korrekt sind, dann sind auch alle anderen, nichtgetesteten ebenfalls korrekt. Die Menge der getesteten Programmläufe muss also irgendwie repräsentativ für die Menge aller denkbaren Programmläufe sein.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Coverage:

- Randfälle
- Bei Verzweigungen: jeder mögliche Pfad

Darauf sind wir unter anderem beim Thema funktionales Programmieren schon herumgeritten: immer die Randfälle mitbedenken.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Randfälle:

- **Beispiel filter:**
  - **Eingabeliste leer**
  - ***Alle* Elemente der nichtleeren Eingabeliste ausgefiltert**
  - ***Kein* Element der nichtleeren Eingabeliste ausgefiltert**
- **Beispiel Dreiecke:**
  - **Alle Ecken auf einer Linie**
  - **Zwei oder drei Ecken aufeinander**

Noch einmal kurz eine Klärung, was mit Randfällen genau gemeint ist.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Randfälle:

- **Beispiel filter:**
  - **Eingabeliste leer**
  - ***Alle* Elemente der nichtleeren Eingabeliste ausgefiltert**
  - ***Kein* Element der nichtleeren Eingabeliste ausgefiltert**
- **Beispiel Dreiecke:**
  - **Alle Ecken auf einer Linie**
  - **Zwei oder drei Ecken aufeinander**

Dieses Beispiel, also was es alles für Randfälle bei der Funktion filter gibt, hatten wir in Kapitel 04b schon ausgiebig diskutiert, genauer: zu einem Spezialfall namens less-than-only. Diese Diskussion müssen wir hier nicht wiederholen.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Randfälle:

- **Beispiel filter:**
  - **Eingabeliste leer**
  - ***Alle* Elemente der nichtleeren Eingabeliste ausgefiltert**
  - ***Kein* Element der nichtleeren Eingabeliste ausgefiltert**
- **Beispiel Dreiecke:**
  - **Alle Ecken auf einer Linie**
  - **Zwei oder drei Ecken aufeinander**

Geometrische Figuren sind auch hier ein dankbares Beispiel, etwa Dreiecke.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Randfälle:

- **Beispiel filter:**
  - **Eingabeliste leer**
  - ***Alle* Elemente der nichtleeren Eingabeliste ausgefiltert**
  - ***Kein* Element der nichtleeren Eingabeliste ausgefiltert**
- **Beispiel Dreiecke:**
  - **Alle Ecken auf einer Linie**
  - **Zwei oder drei Ecken aufeinander**

Der klassische Randfall ist, dass das Dreieck nur eine gerade Strecke ohne Inneres und mit Flächeninhalt 0 ist. Wenn dieser Fall nicht durch eine Vorbedingung ausgeschlossen ist, dann muss auch er korrekt funktionieren.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Randfälle:

- **Beispiel filter:**
  - **Eingabeliste leer**
  - ***Alle* Elemente der nichtleeren Eingabeliste ausgefiltert**
  - ***Kein* Element der nichtleeren Eingabeliste ausgefiltert**
- **Beispiel Dreiecke:**
  - **Alle Ecken auf einer Linie**
  - **Zwei oder drei Ecken aufeinander**

Dieser Randfall lässt sich natürlich noch dahingehend steigern, dass eine Seite Länge 0 hat oder sogar das ganze Dreieck aus nur einem einzigen Punkt besteht. Auch hier gilt: Muss korrekt behandelt werden, wenn nicht durch eine Vorbedingung ausgeschlossen.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

Coverage:

- **Randfälle**
- **Bei Verzweigungen: jeder mögliche Pfad**

Zurück zur Übersicht für Coverage, nun zum zweiten Punkt. Verzweigungen machen Schwierigkeiten, denn bei einer Verzweigung muss jede Möglichkeit überdeckt werden. Das kann ganz schnell zu einer sehr großen Zahl führen.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;
   if ( n % 2 == 1 )
      result – –;
   return result;
}
```

Dazu sehen wir uns ein einfaches Beispiel in zwei Varianten an. Die beiden hier gezeigten Implementationen der rein illustrativen Methode m sollen exakt äquivalent sein.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;
   if ( n % 2 == 1 )
      result – –;
   return result;
}
```

Zwei ineinander geschachtelte if-Abfragen ergeben vier grundsätzlich verschiedene Pfade, über die der Prozess durch die Anweisungen laufen kann, und liefern potentiell vier verschiedene Ergebnisse. Alle vier Fälle müssen durch Laufzeittests überdeckt werden, denn jeder der vier Fälle kann für sich falsch sein, während die drei anderen Fälle korrekt sind. Das Problem ist natürlich, dass sich diese Anzahl mit jeder weiteren Schachtelungstiefe verdoppelt. Glücklicherweise sind tiefe Schachtelungen aber wohl eher selten.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;
   if ( n % 2 == 1 )
      result – –;
   return result;
}
```

Nun schauen wir uns dieselben vier Fälle auf der rechten Seite an, um zu überprüfen, ob die beiden Methoden wirklich in allen vier Fällen dasselbe Ergebnis liefern. Im Falle von positiven ungeraden Zahlen ist das schon einmal so, ...

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;
   if ( n % 2 == 1 )
      result – –;
   return result;
}
```

... ebenso bei positiven geraden Zahlen, ...

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;
   if ( n % 2 == 1 )
      result – –;
   return result;
}
```

„„ bei nichtpositiven ungeraden Zahlen, ...

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;
   if ( n % 2 == 1 )
      result – –;
   return result;
}
```

... und schließlich auch bei nichtpositiven geraden Zahlen. Beide Methoden sind tatsächlich äquivalent.

Laufzeittests

TECHNISCHE UNIVERSITÄT DARMSTADT

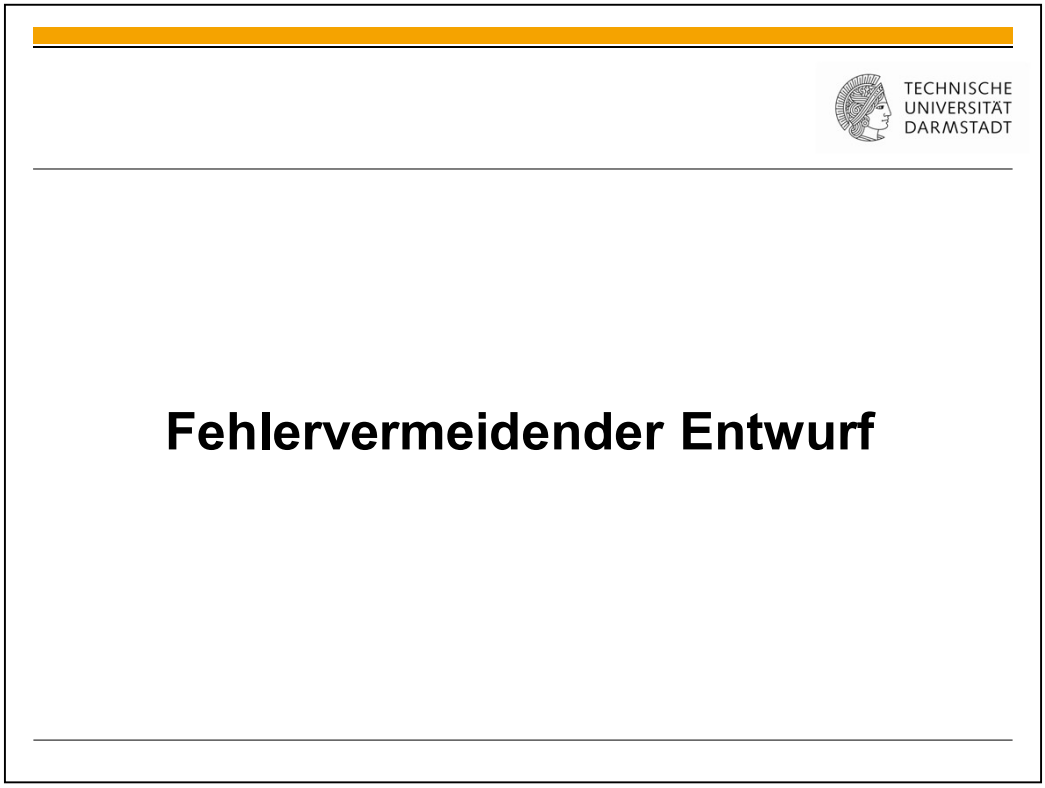
```
public int m ( int n ) {
   if ( n > 0 )
      if ( n % 2 == 1 )
         return 0;
      else
         return 1;
   else if ( n % 2 == 1 )
      return –1;
   return 0;
}
```

```
public int m ( int n ) {
   int result = 0;
   if ( n > 0 )
      result++;     //  !!!
   if ( n % 2 == 1 )
      result – –;   //  !!!
   return result;
}
```

→ n verknüpfte if-Abfragen = 2^n Fälle zu testen !!!

Es geht uns hier aber um etwas anderes: Rechts sind die beiden if-Abfragen nicht ineinander geschachtelt, sondern folgen strikt nacheinander. Sie sind aber ganz und gar nicht unabhängig voneinander, sondern durch die Variable result sogar sehr eng miteinander verknüpft. Ein sehr häufiger Fall, der auch in realistischen Fällen beliebig viele if-Abfragen enthalten kann, nicht nur zwei wie hier. Das hat natürlich zur Konsequenz, dass sehr viele Fälle durch Tests abzudecken sind.

Solche if-Verzweigungen sind in realen Programmen häufig sogar über etliche Methoden verteilt, nicht so schön kompakt wie hier in einer einzigen Methode zusammengefasst, was die ganze Sache natürlich noch unübersichtlicher macht.



Fehlersuche ist gut, Fehler*vermeidung* ist natürlich besser.

Fehlervermeidender Entwurf

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Prinzipien und Techniken für fehler-vermeidenden Entwurf verbessern auch

- **Wartbarkeit**
- **Modifizierbarkeit**
- **Erweiterbarkeit**

Bevor wir einsteigen, sollte erwähnt werden, dass fehlervermeidender Entwurf zugleich auch bedeutet, dass der Quelltext auch besser wartbar sowie in seiner Funktionalität leichter und weniger fehleranfällig modifizierbar und erweiterbar ist. Bis zu einem gewissen Grad sind diese vier Kriterien nur verschiedene Sichtweisen auf dieselbe Problematik.

Fehlervermeidender Entwurf

TECHNISCHE UNIVERSITÄT DARMSTADT

Ausgewählte grundlegende Techniken:

- **KISS**
- **Separation of Concerns**
- **Konformität**

Das sind die Grundsätze, die wir kurz der Reihe nach ansprechen werden. Natürlich wird damit keiner dieser drei Punkte und erst recht nicht das Gesamtthema fehlervermeidender Entwurf auch nur annähernd erschöpfend behandelt sein.

TECHNISCHE UNIVERSITÄT DARMSTADT

KISS

KISS ist die Abkürzung für „keep it simple, stupid!“ Dieses angehängte „stupid“ ist ein gängiges Konstruktionsprinzip für Merksätze im Englischen, man denke etwa auch an den Wahlslogan von Bill Clinton: „it´s the economy, stupid!“

Das KISS-Prinzip besagt, dass man versuchen soll, Quelltexte möglichst einfach und intuitiv zu gestalten und zu strukturieren. Raffinesse in Quelltexten mag dem Ego des Programmierers schmeicheln und seinen Spieltrieb befriedigen, führt aber aller Erfahrung nach zu Quelltext, der erstens unnötig fehlerhaft und zweitens extrem schwer zu durchschauen und daher auch extrem schwer zu warten und weiterzuentwickeln ist.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
while ( *p++ = *q++ );
```

```
while ( *q != 0 ) {
   *p = *q;
   p++;
   q++;
}
*p = 0;
```

Beispiele aus der Sprache C eignen sich besonders gut, um das KISS-Prinzip zu verdeutlichen. Dies ist ein gängiges Beispiel.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
while ( *p++ = *q++ );
```

```
while ( *q != 0 ) {
   *p = *q;
   p++;
   q++;
}
*p = 0;
```

Wie Sie sehen, reicht sogar eine einzige Zeile C-Code zur Illustration aus. Diese Zeile ist äquivalent zu dem Stück C-Code, das Sie auf der rechten Seite der Folie sehen. Die rechte Seite ist sicherlich einfacher durchschaubar; so wie *rechts* sollte es aussehen, nicht so wie *links*!

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
while ( *p++ = *q++ );
```

```
while ( *q != 0 ) {
   *p = *q;
   p++;
   q++;
}
*p = 0;
```

Die Variable q durchläuft die Komponenten eines Strings, der als ganz normales Array von char eingerichtet ist. Mit dem * wird in C auf das Zeichen am aktuellen Index im String zugegriffen. Konvention in C ist, dass ein Array, das einen String darstellt, noch eine weitere Komponente hinter dem Ende des eigentlichen Strings haben muss, in dem das Null-Zeichen steht, so dass Test auf ungleich 0 die korrekte Fortsetzungsbedingung ist.

In dieser Zeile auf der rechten Seite durchschaut man das sofort. Bei der einzelnen Zeile auf der linken Seite muss man erst einmal nachvollziehen, dass die Fortsetzungsbedingung der while-Schleife tatsächlich ein boolescher Ausdruck ist, dass der Wert dieses Ausdrucks gleich dem Wert von *q ist und dass das Null-Zeichen in der Fortsetzungsbedingung als false, jedes andere Zeichen als true interpretiert wird.

***Erinnerung*: In Kapitel 01b, Abschnitt Bindungsstärke von Operatoren, hatten wir festgestellt, dass der Zuweisungsoperator einen Rückgabewert hat, nämlich den kopierten Wert. Das ist in C genauso.**

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
while ( *p++ = *q++ );
```

```
while ( *q != 0 ) {
  *p = *q;
  p++;
  q++;
}
*p = 0;
```

Auf der rechten Seite durchschaut man außerdem sofort, was die Schleife eigentlich in jedem Durchlauf tun soll. An dem Index, auf dem die beiden Variablen in ihren beiden Strings stehen, soll das Zeichen von dem einen in den anderen String kopiert werden. Danach gehen beide Laufvariablen einen Schritt weiter in ihrem jeweiligen String, also zum nächsten Index. Insbesondere ist rechts offensichtlich, dass auch an dem Index, mit dem p und q vor der Schleife initialisiert wurden, eine Kopieraktion stattfindet.

Auf der linken Seite muss man sich erst klarmachen, dass das ++ zwar höhere Bindungsstärke als der Stern hat, so dass ++ auf die Laufvariablen p und q und nicht auf die Zeichen *p beziehungsweise *q angewandt wird, aber als Postfix-Operator erst ganz am Ende ausgeführt wird, so dass also links die Operatoren in derselben Reihenfolge ausgeführt werden wie rechts.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
while ( *p++ = *q++ );
```

```
while ( *q != 0 ) {
   *p = *q;
   p++;
   q++;
}
*p = 0;
```

Schließlich muss man sich links noch klarmachen, dass tatsächlich auch das Null-Wort mitkopiert wird, weil die Fortsetzungsbedingung der while-Schleife erst ausgewertet wird, *nachdem* das Zeichen kopiert wurde. Auch dieser Umstand ist rechts offensichtlich beziehungsweise man würde das Fehlen der letzten Zeile beim Durchdenken der Schleife sicherlich schnell bemerken.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
   case 0: do { *p++ = *q++;
   case 7:      *p++ = *q++;
   case 6:      *p++ = *q++;
   case 5:      *p++ = *q++;
   case 4:      *p++ = *q++;
   case 3:      *p++ = *q++;
   case 2:      *p++ = *q++;
   case 1:      *p++ = *q++;
} while ( – – n  > 0 ); }
```

Einen besonders raffinierten Verstoß gegen das KISS-Prinzip in der Sprache C möchte der Autor dieser Folien Ihnen nicht vorenthalten. Dieses Stück Code ist bekannt geworden und im Internet leicht auffindbar unter dem Namen Duff‘s Device, benannt nach seinem Erfinder, Tom Duff.

***Erinnerung*: Wir hatten dieses Beispiel schon einmal in Kapitel 13 unter dem Stichwort Unrolling gesehen.**

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
    case 0: do { *p++ = *q++;
    case 7:      *p++ = *q++;
    case 6:      *p++ = *q++;
    case 5:      *p++ = *q++;
    case 4:      *p++ = *q++;
    case 3:      *p++ = *q++;
    case 2:      *p++ = *q++;
    case 1:      *p++ = *q++;
  } while ( – – n  > 0 ); }
```

Wie Sie sehen, kann man in C eine switch-Anweisung und eine do-while-Schleife so ineinander verschachteln, dass das do innerhalb und das while außerhalb der switch-Anweisung ist.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
   case 0: do { *p++ = *q++;
   case 7:      *p++ = *q++;
   case 6:      *p++ = *q++;
   case 5:      *p++ = *q++;
   case 4:      *p++ = *q++;
   case 3:      *p++ = *q++;
   case 2:      *p++ = *q++;
   case 1:      *p++ = *q++;
} while ( – – n  > 0 ); }
```

Um Laufzeit zu sparen, soll die Fortsetzungsbedingung der do-while-Schleife nicht bei jedem Kopiervorgang geprüft werden, sondern nur bei jedem achten. Dafür steht achtmal dieselbe Zuweisung im Schleifenrumpf – also tatsächlich ein Beispiel für Unrolling.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
    case 0: do { *p++ = *q++;
    case 7:      *p++ = *q++;
    case 6:      *p++ = *q++;
    case 5:      *p++ = *q++;
    case 4:      *p++ = *q++;
    case 3:      *p++ = *q++;
    case 2:      *p++ = *q++;
    case 1:      *p++ = *q++;
} while ( – – n  > 0 ); }
```

Die Anzahl der Durchläufe muss dementsprechend um den Faktor 8 reduziert werden. In count steht die ursprüngliche Anzahl, n enthält die um Faktor 8 reduzierte Anzahl.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
  case 0: do { *p++ = *q++;
  case 7:      *p++ = *q++;
  case 6:      *p++ = *q++;
  case 5:      *p++ = *q++;
  case 4:      *p++ = *q++;
  case 3:      *p++ = *q++;
  case 2:      *p++ = *q++;
  case 1:      *p++ = *q++;
} while ( – – n  > 0 ); }
```

Nun ist die Gesamtzahl der Durchläufe aber nicht unbedingt durch 8 teilbar, es bleibt ein Rest. Dafür ist die switch-Anweisung da. Die Zahl n enthält nicht genau die eigentliche Anzahl der Durchläufe geteilt durch 8, sondern dieses Ergebnis wird aufgerundet. Dafür sorgt die Addition von 7 auf count, bevor ganzzahlige Division durch 8 angewendet wird. Je nachdem, was der Rest von count geteilt durch 8 ist, springt das Programm genau an die Stelle in der Schleife, so dass bis zur Fortsetzungsbedingung genau so viele Zuweisungen ausgeführt werden, wie der Rest der Division von count durch 8 besagt.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

In der Spache C:

```
int n = ( count + 7 ) / 8;
switch ( count % 8 ) {
   case 0: do { *p++ = *q++;
   case 7:      *p++ = *q++;
   case 6:      *p++ = *q++;
   case 5:      *p++ = *q++;
   case 4:      *p++ = *q++;
   case 3:      *p++ = *q++;
   case 2:      *p++ = *q++;
   case 1:      *p++ = *q++;
} while ( – – n  > 0 ); }
```

Duff selbst hat seine Erfindung übrigens sinngemäß so kommentiert: Ihm ist klar, dass dieses Beispiel ein starkes Argument ist, aber ihm ist nicht klar, ob es nun ein Pro- oder ein Kontra-Argument ist.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

- Dekomposition in kleine, überschaubare Einheiten.
  - ➢Einheiten: Klassen, Subroutinen etc.
- Programm soll nicht raffiniert, sondern in intuitiv sofort verständliche, realitätsabbildende Einheiten zerlegt werden.
- Die Identifier für diese Einheiten sollen die Intuition unterstützen.

Weiter mit allgemeinen Betrachtungen zum KISS-Prinzip: Zum KISS-Prinzip gehört auch, dass das Gesamtprogramm nicht als ein großer Monolith realisiert, sondern in viele kleine Einheiten zerlegt werden sollte, ...

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Dekomposition in kleine, überschaubare Einheiten.**
  - ➢ **Einheiten: Klassen, Subroutinen etc.**
- **Programm soll nicht raffiniert, sondern in intuitiv sofort verständliche, realitätsabbildende Einheiten zerlegt werden.**
- **Die Identifier für diese Einheiten sollen die Intuition unterstützen.**

... die jeweils für sich gut verständlich und überschaubar klein sind.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

- Dekomposition in kleine, überschaubare Einheiten.
  - ➢Einheiten: Klassen, Subroutinen etc.
- Programm soll nicht raffiniert, sondern in intuitiv sofort verständliche, realitätsabbildende Einheiten zerlegt werden.
- Die Identifier für diese Einheiten sollen die Intuition unterstützen.

Wir hatten schon mehrfach gesagt, was wir unter solchen Einheiten verstehen.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

- Dekomposition in kleine, überschaubare Einheiten.
  - ➢Einheiten: Klassen, Subroutinen etc.
- Programm soll nicht raffiniert, sondern in intuitiv sofort verständliche, realitätsabbildende Einheiten zerlegt werden.
- Die Identifier für diese Einheiten sollen die Intuition unterstützen.

Auch hierbei ist es wieder wichtig, möglichst intuitiv verständlichen Code zu schreiben.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

- Dekomposition in kleine, überschaubare Einheiten.
  - ➢Einheiten: Klassen, Subroutinen etc.
- Programm soll nicht raffiniert, sondern in intuitiv sofort verständliche, realitätsabbildende Einheiten zerlegt werden.
- Die Identifier für diese Einheiten sollen die Intuition unterstützen.

Je weniger künstlich die Einheiten gebildet werden, um so verständlicher ist natürlich das Gesamtprogramm.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

- Dekomposition in kleine, überschaubare Einheiten.
  - ➢Einheiten: Klassen, Subroutinen etc.
- Programm soll nicht raffiniert, sondern in intuitiv sofort verständliche, realitätsabbildende Einheiten zerlegt werden.
- Die Identifier für diese Einheiten sollen die Intuition unterstützen.

Und natürlich sollte man dabei auch an die Namensgebung denken.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel:

```
( define ( tax-debt person ) ( ..... ) )
( define ( assessed-tax person ) ( ..... ) )
( define ( pre-payment person ) ( ..... ) )
( define ( tax-rate income ) ( ..... ) )
( define ( annual-salary person ) ( ..... ) )
( define ( additional-income person ) ( ..... ) )
( define tax-exempt-amount ) ( ..... ) )
( define tax-free-amount ( ..... ) )
```

***Erinnerung*: Dies sind ein paar der Funktionen aus dem Abschnitt zur schrittweisen Verfeinerung in Kapitel 04d.**

Die Zerlegung der Gesamtaufgabe in einzelne Einheiten und die Namensgebung sind sicherlich ausreichend intuitiv, um auch von Leuten sofort verstanden zu werden, die bislang mit dem Programm noch gar nichts zu tun hatten – im Idealfall zumindest in groben Zügen sogar von Leuten, die gar nicht programmieren können.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel aus der Java-Standardbibliothek:

```
public class Component { ..... }
public class Button extends Component { ..... }
public class Canvas extends Component { ..... }
public class TextComponent extends Component { ..... }
public class TextField extends TextComponent { ..... }
public class TextArea extends TextComponent { ..... }
```

Als nächstes ein Beispiel aus Java, diesmal nicht selbst geschriebene Einheiten, sondern Klassen aus der Standardbibliothek.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel aus der Java-Standardbibliothek:

```
public class Component { ..... }
public class Button extends Component { ..... }
public class Canvas extends Component { ..... }
public class TextComponent extends Component { ..... }
public class TextField extends TextComponent { ..... }
public class TextArea extends TextComponent { ..... }
```

Die Klasse Component ist die natürliche Abstraktion für alle Arten von GUI-Komponenten. Es unterstützt daher die Intuition, wenn man eine Klasse namens Component definiert, die die gemeinsame Basisklasse aller Arten von GUI-Komponenten ist.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel aus der Java-Standardbibliothek:

```
public class Component { ..... }
public class Button extends Component { ..... }
public class Canvas extends Component { ..... }
public class TextComponent extends Component { ..... }
public class TextField extends TextComponent { ..... }
public class TextArea extends TextComponent { ..... }
```

Ebenso sind sich TextField und TextArea in der menschlichen Betrachtung so ähnlich, dass eine gemeinsame Abstraktion Sinn macht und der Name TextComponent auch wieder intuitiv ist.

Man hätte sich statt dessen auch überlegen können, beispielsweise für Canvas und TextArea oder für TextField und Label eine gemeinsame Abstraktion zu definieren. Möglicherweise hätte das Code gespart, aber es wäre wohl nicht so intuitiv gewesen und hätte damit gegen das KISS-Prinzip verstoßen.

***Erinnerung*: Einen Verstoß gegen das KISS-Prinzip hatten wir im Abschnitt zu Swing in Kapitel 10 gesehen: Klasse JComponent war aus Gründen, die dort diskutiert wurden, nicht von Klasse Component, sondern von Klasse Container abgeleitet worden, was sicherlich eher unintuitiv ist.**

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

Es gibt einfache Regeln, mit denen man die Verständlichkeit von Einheiten mittels Namensgebung deutlich verbessern kann.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

Objekte stellen in der Regel Entitäten da, daher sollten sie selbst und ihre Typen mit Substantiven bezeichnet werden.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

Eine boolesche Variable, eine boolesche Konstante, ein boolescher Parameter, eine Subroutine mit boolescher Rückgabe, alles das realisiert Prädikate und sollte entsprechend benannt sein.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

Variablen, Konstanten, Parameter von anderem als booleschem Typ, Subroutinen mit anderem als booleschem Rückgabetyp – in solchen Fällen kann man den Wert meist mit einem Nomen umschreiben, also mit einem Substantiv wie in den Beispielen auf dieser Folie; zuweilen auch mit einem Adjektiv, zum Beispiel Color.RED.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

Eine void-Methode sowie vergleichbare Konstrukte in anderen Programmiersprachen sollen ja nichts zurückliefern, sondern etwas tun. Dafür ist dann ein Imperativ als Name intuitiv.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

Wir hatten schon mehrfach eine Konvention gesehen, die davon etwas abweicht: Methoden, mit denen man lesend oder schreibend auf ein Attribut zugreift, werden speziell in Java aus get beziehungsweise set und dem Attributnamen gebildet. Bei der set-Methode passt das zur vorherigen Regel, dass der Name einer void-Methode ein Imperativ sein sollte; bei der get-Methode passt die Regel weiter oben *nicht*, dass der Name einfach nur eine Beschreibung des Wertes ist.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Objekte bzw. Klassen / Interfaces: Substantive**
  - **Robot, myRobot, String, Matrix, Button, GeomShape2D, Stream .....**
- **Boolesche Bestandteile: Prädikate**
  - **isGreen, hasButton, thisDamnedPieceOfCodeStillProducesErrors .....**
- **Andere Bestandteile mit Wert: Beschreibung des Wertes**
  - **tax-debt, pre-payment, annual-salary, tax-exempt-amount .....**
- **Subroutinen ohne Rückgabe: Imperativ**
  - **computeTax, fillOval, print, close .....**
- **Ausnahme (reale und virtuelle) Attribute: get/set + Attributnamen**

***Erinnerung*: In Kapitel 02 und anderswo hatten wir auch Fälle gesehen, in denen das Attribut, auf das mit get- und set-Methode zugegriffen wird, gar nicht physisch existiert, also virtuell aus anderen Daten gebildet wird.**

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Zum Abschluss dieses Abschnitts noch ein paar allgemeinere Überlegungen zu Namenskonventionen.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Sie haben ja immer wieder gesehen, dass Identifier typischerweise aus mehreren Wortbestandteilen zusammengesetzt sind. Das ist wichtig, denn jeder Aspekt, der nicht im Identifier berücksichtigt wurde, kann leicht bei der Verwendung übersehen werden.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel:

```
public void closeAll () {
   ab.close();
   bc.close();
   cd.close();
   System.exit ( 0 );
}
```

An dieser Stelle der allgemeinen Überlegungen schwenken wir kurz um auf ein leider gar nicht so ganz seltenes Beispiel, wie man es nicht machen sollte beziehungsweise wie man es besser machen kann.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel:

```
public void closeAll () {
  ab.close();
  bc.close();
  cd.close();
  System.exit ( 0 );
}
```

Wie der Name dieser Methode schon sagt, soll sie diverse Objekte schließen, wobei es uns für dieses Beispiel egal ist, was das für Objekte sind; so weit, so gut.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel:

```
public void closeAll () {
   ab.close();
   bc.close();
   cd.close();
   System.exit ( 0 );
}
```

Aus unerfindlichen Gründen hat der Programmierer dieser Methode aber entschieden, dass mit dem Schließen der Geräte auch der gesamte Programmlauf beendet werden soll. Kann man natürlich so machen, ist vom Programmierer vielleicht auch sorgfältig in der Dokumentation der Methode closeAll vermerkt worden. Aber der Name der Methode sollte unbedingt diesen ja nun nicht gerade vernachlässigbaren Umstand ebenfalls reflektieren.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel:

```
public void closeAllAndExit () {
   ab.close();
   bc.close();
   cd.close();
   System.exit ( 0 );
}
```

Zum Beispiel einfach so: Beide Funktionalitäten dieser Methode sind nun im Namen der Methode genannt, man muss nicht erst die Dokumentation der Methode nachschlagen, um eine Überraschung zu vermeiden.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Denn bedenken Sie: Mit dem Namen einer Entität muss man zwangsweise umgehen, wenn man die Entität verwenden will. Bei jeder Verwendung erst die Doku zur Entität nachzuschlagen, ist hingegen mühselig, und wenn man keine Überraschungen erwartet, dann wird man das tendenziell unterlassen – und gegebenenfalls hereinfallen.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Sie werden einwerfen, dass Identifier dadurch zu lang werden, wenn man wirklich jeden relevanten Aspekt in den Identifier aufnehmen will. Mein Gegenargument ist, dass dies ein deutliches Anzeichen dafür ist, dass die Struktur des Programms immer noch nicht ausreichend konsequent in übersichtliche, intuitive Einheiten aufgegliedert ist.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Natürlich gibt es keine Regeln ohne Ausnahmen.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Zum Beispiel wird man die Namen von mathematischen Variablen oder von Laufindizes nicht mit aller Gewalt aussagekräftig aus ausgeschriebenen Wörtern zusammensetzen, denn man kann sicherlich davon ausgehen, dass alle, die mit dem Quelltext arbeiten sollen, mit den typischen Notationen aus der Mathematik vertraut sind und diese auch als intuitiv und leicht verständlich empfinden werden.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Bei den Namen von Packages haben sich Kurzformen und Abkürzungen als Standard etabliert, zum Beispiel lang für language, util für utilities oder auch awt für abstract window toolkit.

KISS

TECHNISCHE UNIVERSITÄT DARMSTADT

Namenskonventionen:

- **Generelle Regel:**
  - **Alle wichtigen Aspekte zu Wortbestandteilen machen.**
  - **Identifier ist der *einzige* Kommentar, der bei jeder Nutzung dabei steht!**
  - **Zu viele wichtige Aspekte in einem Identifier → dringend Struktur gemäß KISS überdenken.**
- **Ausnahmen:**
  - **Typische mathematische Notation wie i, j, x, y, x1, x2, x3 .....**
  - **Packagenamen eher kurz.**
  - **im Kontext allgemein als bekannt voraussetzbare Abkürzungen wie FRA, TXL, HHN, FKB, MUC ..... beim Kontext Flughäfen.**

Und natürlich kann man auch auf die Begrifflichkeiten und Notationen aus dem jeweiligen Anwendungskontext zurückgreifen, um eher kurze Identifier zu wählen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Separation of Concerns (SoC)

Zum nächsten ausgewählten Ansatz für fehlervermeidenden Entwurf. Der als nächstes zu besprechende, extrem wichtige Grundsatz ist uns schon des Öfteren begegnet, ohne dass wir ihn benannt haben. Jetzt betrachten wir ihn systematisch.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel in Racket:

```
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Ein sehr schönes Beispiel dazu hatten wir schon in Kapitel 04c gesehen, Abschnitt zu Kombinationen aus fold, filter und map. Hier ist dasselbe Beispiel nochmals zu sehen.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel in Racket:

```
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Eine etwas komplexere Aufgabe ist zu lösen: Die Liste enthält nicht nur Studierende, aber von allen Studierenden in der Liste sollen die Gebühren aufsummiert werden.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel in Racket:

```
( define ( sum-of-student-fees list )
  ( my-fold
    ( lambda ( x y ) ( + x y ) )
    0
    ( my-map
      ( lambda ( stud ) ( student-fee stud ) )
      ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Die Lösung der Aufgabe enthält drei sehr allgemeine, abstrakte Aspekte. Im Hinblick auf das Thema des Abschnitts, Separation of Concerns, kann man Aspekte vielleicht mit Concerns übersetzen. Für diese drei Aspekte oder Concerns hatten wir jeweils eine eigene Funktion definiert, die den jeweiligen Aspekt und nichts darüber hinaus realisiert.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel in Racket:

```
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
       ( lambda ( stud ) ( student-fee stud ) )
       ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Zu den drei abstrakten, allgemeinen Aspekten gehört jeweils ein konkreter Aspekt, der sich in einem der Parameter ausdrückt. Diese drei Parameter hatten wir im Beispiel als Lambda-Ausdrücke realisiert.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel in Racket:

```
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Genau das ist Separation of Concerns: Die Gesamtaufgabe wurde in verschiedene Aspekte zerlegt, in diesem Fall in sechs. Für einige dieser Aspekte gibt es schon Lösungen, weil sie eben sehr häufig sind, das sind im Beispiel die Funktionalitäten von filter, fold und map. Die anderen drei Aspekte sind aufgabenspezifisch und werden ad-hoc dazu geschrieben, in diesem Beispiel, wie gesagt, in Form von Lambda-Ausdrücken.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele in der Java-Standardbibliothek:

- **Runnable vs. Thread**
- **Component vs. Listener vs. Event**
- **Collections.sort vs. Comparator**
- **InputStream vs. Reader bzw. OutputStream vs. Writer**
- **Reader, BufferedReader, LineNumberReader .....**

In Java ist uns das Konzept Separation of Concerns an vielen Stellen begegnet, insbesondere in der Standardbibliothek. Wir gehen hier nur ein paar Beispiele schnell durch, das sollte reichen, um das Konzept und seine Umsetzung in Java zu verstehen.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele in der Java-Standardbibliothek:

- Runnable vs. Thread
- Component vs. Listener vs. Event
- Collections.sort vs. Comparator
- InputStream vs. Reader bzw. OutputStream vs. Writer
- Reader, BufferedReader, LineNumberReader .....

***Erinnerung*: In Kapitel 09 hatten wir gesehen, dass die Funktionalität von Threads in zwei Concerns zerlegt ist. Die spezifische Logik des Threads steckt in einer individuell geschriebenen Klasse, die von Interface Runnable abgeleitet ist. Die technische Umsetzung von Threads – also das, was allen Threads gemeinsam ist – steckt in der Klasse Thread.**

Genau diese Konstellation hatten wir eben auch im Racket-Beispiel: Der eine Concern ist allgemein, der andere Concern ist aufgabenspezifisch.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele in der Java-Standardbibliothek:

- **Runnable vs. Thread**
- **Component vs. Listener vs. Event**
- **Collections.sort vs. Comparator**
- **InputStream vs. Reader bzw. OutputStream vs. Writer**
- **Reader, BufferedReader, LineNumberReader .....**

***Erinnerung*: In Kapitel 10 hatten wir gesehen, dass ein wichtiges Concern der Klasse Component ausgelagert ist, nämlich die Fähigkeit, auf Nutzereingaben per Maus oder Tastatur zu reagieren. Dafür sind die Listener-Interfaces bereitgestellt. Auch hier gibt es noch einen Fall von Separation of Concerns: Die eigentliche Mitprotokollierung der Nutzereingabe ist nicht innerhalb der Listener fest verdrahtet, sondern in die Event-Klassen ausgelagert.**

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele in der Java-Standardbibliothek:

- Runnable vs. Thread
- Component vs. Listener vs. Event
- Collections.sort vs. Comparator
- InputStream vs. Reader bzw. OutputStream vs. Writer
- Reader, BufferedReader, LineNumberReader .....

***Erinnerung*: In Kapitel 06 gibt es einen Abschnitt zum Interface Comparator. Dieser steht für einen gewissen Concern, der in Comparator ausgelagert ist, nämlich wie die einzelnen Werte des generischen Typs in eine Reihung zu bringen sind – was ja beim selben Typ durchaus in unterschiedlicher Weise geschehen kann, wie wir dort gesehen hatten.**

Sortieralgorithmus und Reihungslogik sind also zwei verschiedene Concerns, die im Design der Klassenmethode sort von Klasse Collections voneinander separiert sind, und der zweite Concern kommt dann als Parameter der Methode sort in den ersten Concern wieder hinein.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele in der Java-Standardbibliothek:

- **Runnable vs. Thread**
- **Component vs. Listener vs. Event**
- **Collections.sort vs. Comparator**
- **InputStream vs. Reader bzw. OutputStream vs. Writer**
- **Reader, BufferedReader, LineNumberReader .....**

Bei Weitem nicht in jeder Programmiersprache gibt es diese Unterscheidung zwischen dem byteweisen und dem zeichenweisen Lesen und Schreiben von Daten. Dieses Beispiel für Separation of Concerns sollte man daher nicht für selbstverständlich nehmen.

***Erinnerung*: Die Referenztypen in diesem und dem nächsten Punkt haben Sie in Kapitel 08 gesehen.**

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele in der Java-Standardbibliothek:

- Runnable vs. Thread
- Component vs. Listener vs. Event
- Collections.sort vs. Comparator
- InputStream vs. Reader bzw. OutputStream vs. Writer
- Reader, BufferedReader, LineNumberReader .....

Auch dieses Beispiel von Separation of Concerns ist nicht selbstverständlich: Ob man Eingaben puffert oder nicht, ob man zeilenweise einlesen kann oder nicht, und so weiter, das lässt sich in Java alles beliebig kombinieren, indem man einen Reader als Parameter in den Konstruktor des anderen steckt.

Das sind alles unterschiedliche Concerns, die in Java beliebig kombinierbar sind, wie wir es in Kapitel 08 gesehen hatten.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Sinn von SoC:

- **Übersichtlichere Programmstruktur**
- **Wiederverwendbarkeit**
- **Austauschbarkeit**
  - **Selektiv und unabhängig**

Anhand all dieser Beispiele sollten die Vorteile des Konzepts Separation of Concerns deutlich geworden sein.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Sinn von SoC:

- Übersichtlichere Programmstruktur
- Wiederverwendbarkeit
- Austauschbarkeit
  - Selektiv und unabhängig

Zunächst einmal mag die Programmstruktur komplizierter erscheinen dadurch, dass viele kleine Klassen für die verschiedenen Concerns zu entwickeln und miteinander zu verknüpfen sind. Aber wenn diese Concerns nach dem KISS-Prinzip einfach und intuitiv gewählt worden sind, sollte der Quelltext und seine Funktionsweise unterm Strich *besser* zu durchschauen sein.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Sinn von SoC:

- Übersichtlichere Programmstruktur
- Wiederverwendbarkeit
- Austauschbarkeit
  - Selektiv und unabhängig

So ziemlich jedes Beispiel auf den letzten Folien sollte gezeigt haben, auf welche Weise Separation of Concerns die Wiederverwendbarkeit von Code unterstützt: Die zu lösende Aufgabe wird in einen oder mehrere allgemeine und einen oder mehrere anwendungsspezifische Aspekte zerlegt, und die allgemeinen Aspekte werden als Klassen oder Subroutinen realisiert, die dann auch in anderen Kontexten verwendbar sind, in denen die spezifischen Aspekte anders definiert sind.

Wie die bisherigen Beispiele immer wieder gezeigt haben, sind die allgemeinen Aspekte sehr häufig schon in der Standardbibliothek der jeweiligen Programmiersprache vorhanden und müssen daher nicht selbst entwickeln werden.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Sinn von SoC:

- Übersichtlichere Programmstruktur
- Wiederverwendbarkeit
- Austauschbarkeit
  - Selektiv und unabhängig

Die Wiederverwendbarkeit beruht im Wesentlichen auf der Austauschbarkeit. Beispielsweise in den Funktionen fold, filter und map sind die Funktionsparameter austauschbar, in Klasse Thread ist die Runnable-Klasse austauschbar, in Methode sort von Collections ist der Comparator austauschbar, und so weiter.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

Sinn von SoC:

- Übersichtlichere Programmstruktur
- Wiederverwendbarkeit
- Austauschbarkeit
  - ➢ Selektiv und unabhängig

Der wesentliche Punkt ist, dass die einzelnen Concerns zumindest teilweise unabhängig von den anderen Concerns ausgetauscht werden können. Das Beispiel Reader und Writer mag unter allen bisher betrachteten Beispielen dem Idealfall am nächsten kommen: Ob man beispielsweise eine bestimmte Funktionalität wie Pufferung der Eingabe hat oder nicht, ist unabhängig davon, ob man zeilenweises Einlesen hat oder nicht, und so weiter.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Person { ..... }

public class Employee extends Person {
   private String company;
   .....
}

public class Student extends Person {
   private String enrollmentNumber;
   .....
}
```

Wir kommen zu einem Beispiel für Designprobleme, die auf unzureichender Separation of Concerns beruhen. Sowohl dieses Beispiel als auch analoge Beispiele werden häufig in Lehrbüchern und Skripten als beispielhaft *positives* Design präsentiert. Ich hoffe, ich werde Sie überzeugen können, dass dies leider ein *negatives* Beispiel ist.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Person { ..... }

public class Employee extends Person {
    private String company;
    .....
}

public class Student extends Person {
    private String enrollmentNumber;
    .....
}
```

Wenn man für Personen allgemein sowie für Beschäftigte und Studierende jeweils eine Klasse haben möchte, dann bietet es sich an, die beiden letzteren Klassen von der ersteren abzuleiten, denn jeder Beschäftigte und jeder Studierende ist ja auch eine Person.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Person { ..... }

public class Employee extends Person {
   private String company;
   .....
}

public class Student extends Person {
   private String enrollmentNumber;
   .....
}
```

In Klasse Person würde man allgemeine Attribute von Personen wie Vorname, Nachname, Geburtsdatum und Anschrift verwalten. Durch die Ableitungsbeziehung hätte jeder Beschäftigte und jeder Studierende diese Attribute ebenfalls. Zusätzlich kann man spezifische Attribute in die abgeleiteten Klassen einfügen, zum Beispiel Firmenname und Abteilungsname bei einem Beschäftigten und die Matrikelnummer bei einem Studierenden. Sieht also auf den ersten Blick wie die perfekte Lösung aus, richtig?

Leider doch nicht so recht, denn dieses Klassendesign berücksichtigt nicht den Fall, dass dieselbe Person bei mehreren Firmen gleichzeitig arbeiten und in mehreren Hochschulen gleichzeitig eingeschrieben und sogar in einer Firma beschäftigt sein und gleichzeitig an einer Hochschule studieren kann. Um dies zu erreichen, bräuchte es für jede dieser Rollen ein eigenes Objekt, und die Personendaten aller dieser Objekte sind dieselben, es handelt sich ja um dieselbe Person. Eine solche unkontrollierte Redundanz ist nicht nur hochgradig unschön, sie ist auch eine virulente Fehlerquelle.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Employee {
   private Person person;
   .....
   public Employee ( Person person, String company ) { ..... }
   public Person getPerson () {
      return person;
   }
}

Employee employee = new Employee ( person, “TU Darmstadt“ );
String lastName = employee.getPerson().getLastName();
```

Das entscheidende Wort ist eben schon gefallen: „Rolle“. Bei einer Firma beschäftigt zu sein beziehungsweise an einer Hochschule zu studieren, das sind *Rollen*, die eine Person *einnimmt*. Diese Erkenntnis sollte der Ausgangspunkt des Designs sein. Eine beispielhafte Umsetzung sehen Sie hier, nur für Beschäftigte, analog andere Rollen wie Studierende.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Employee {
   private Person person;
   .........
   public Employee ( Person person, String company ) { ..... }
   public Person getPerson () {
      return person;
   }
}

Employee employee = new Employee ( person, “TU Darmstadt“ );
String lastName = employee.getPerson().getLastName();
```

Die Klasse Person ist in diesem Design nicht die Basisklasse für die Klasse Employee, sondern Employee verweist auf ein Objekt von Klasse Person, das im Konstruktor übergeben wird. Der entscheidende Punkt ist: Mehrere Objekte von Klassen wie Employee oder Student können auf dasselbe Objekt von Klasse Person verweisen. Die Logik ist, dass jedes Objekt von Employee beziehungsweise Student, das auf ein bestimmtes Person-Objekt verweist, eben eine *Rolle* dieser Person repräsentiert.

Separation of Concerns

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Employee {
   private Person person;
   .....
   public Employee ( Person person, String company ) { ..... }
   public Person getPerson () {
      return person;
   }
}

Employee employee = new Employee ( person, "TU Darmstadt" );
String lastName = employee.getPerson().getLastName();
```

Die Person zu einer Rolle kann dann von dem Rollenobjekt mit einer get-Methode abgefragt werden. Will man etwa den Nachnamen eines Beschäftigten haben, muss man so wie hier zugreifen, wobei hier angenommen wird, dass Klasse Person eine get-Methode für den Nachnamen hat.

TECHNISCHE UNIVERSITÄT DARMSTADT

Model – View – Controller

(MVC)

Für das wohl wichtigste und häufigste Beispiel von Separation of Concerns hat sich das Kürzel MVC für Model-View-Controller etabliert. Es geht um Programme mit starkem GUI-Bezug, was heutzutage ja eher der Normalfall ist.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Model-View-Controller (MVC):

- ***Model*: die eigentliche Logik des Programms.**
- ***View* (*Präsentation*):**
  - **Darstellung der Modelldaten,**
  - **Interaktion mit dem Nutzer.**
- ***Controller* (*Steuerung*): Verwaltung des Programmlaufs.**
  - **Häufig eher klein („Geschäftslogik“ oft im Model).**

Erst einmal eine einführende Übersicht, worum es bei MVC eigentlich geht, danach dann zwei Beispiele und die Diskussion der Vorteile von MVC.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Model-View-Controller (MVC):

- ***Model*: die eigentliche Logik des Programms.**
- ***View* (*Präsentation*):**
  - **Darstellung der Modelldaten,**
  - **Interaktion mit dem Nutzer.**
- ***Controller* (*Steuerung*): Verwaltung des Programmlaufs.**
  - **Häufig eher klein („Geschäftslogik“ oft im Model).**

Vor allem geht es darum, die Programmteile, die die eigentliche Logik des Programms realisieren, freizuhalten von allem, was damit *nicht* zu tun hat, insbesondere mit allem, was nur mit der *Darstellung* der Logik auf den Ausgabegeräten und mit Nutzerinteraktionen zu tun hat.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Model-View-Controller (MVC):

- ***Model***: **die eigentliche Logik des Programms.**
- ***View*** **(*Präsentation*):**
  - **Darstellung der Modelldaten,**
  - **Interaktion mit dem Nutzer.**
- ***Controller*** **(*Steuerung*): Verwaltung des Programmlaufs.**
  - **Häufig eher klein („Geschäftslogik“ oft im Model).**

Diese Aspekte sollen dann in einem separaten, möglichst unabhängigen Teil des Gesamtprogramms zusammengefasst werden, eben dem View.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Model-View-Controller (MVC):

- ***Model*: die eigentliche Logik des Programms.**
- ***View* (*Präsentation*):**
  - **Darstellung der Modelldaten,**
  - **Interaktion mit dem Nutzer.**
- ***Controller* (*Steuerung*): Verwaltung des Programmlaufs.**
  - **Häufig eher klein („Geschäftslogik“ oft im Model).**

Natürlich muss es dann noch eine übergeordnete Instanz zur Steuerung geben.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Model-View-Controller (MVC):

- ***Model*: die eigentliche Logik des Programms.**
- ***View* (*Präsentation*):**
  - **Darstellung der Modelldaten,**
  - **Interaktion mit dem Nutzer.**
- ***Controller* (*Steuerung*): Verwaltung des Programmlaufs.**
  - **Häufig eher klein („Geschäftslogik“ oft im Model).**

Die ursprüngliche Idee von MVC war, alles, was sich unter dem Begriff Geschäftslogik subsumieren lässt, in den Controller zu packen, was dem Controller in der Regel eine stattliche Größe gibt. Aber oft passt der Großteil der Geschäftslogik inhaltlich doch eher zum Model, so dass der Controller häufig nur noch gewisse Kernfunktionalitäten hat.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Wir schauen uns MVC anhand von zwei konkreten Beispielen genauer an. Das erste Beispiel ist ein Programm für ein Brettspiel, zum Beispiel Schach – natürlich stark vereinfacht, denn es geht hier nur ums Prinzip.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Das Model enthält alles, was zur Verwaltung eines aktuell laufenden Schachspiels notwendig ist. Das ist sehr wenig, im Grunde nur die aktuelle Stellung, also welche Figur auf welchem Feld steht. Dafür reicht sogar eine Matrix von Feldern aus, deren Komponententyp die Figurenart und ihre Spielfarbe kodiert, zum Beispiel eine Zahl zur Identifizierung der Figurenart und ein boolescher Wert zur Unterscheidung von schwarz und weiß.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Irgendwelche weiteren Informationen, die nichts mit der GUI zu tun haben, beispielsweise Vorwissen über den Spielstil des aktuellen Nutzers, gehören ebenfalls ins Model.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Der Kern der Spiellogik ist natürlich die Subroutine, die auf Basis der momentanen Stellung – gegebenenfalls unter Hinzuziehung von Vorwissen über den Spieler – den nächsten Zug berechnet.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Die GUI-Darstellung des Modells ist eine der Kernaufgaben des Views. Im View eingekapselt sind alle visuellen Informationen, also zum Beispiel, wie das Brett, die 64 Felder und die Spielfiguren dargestellt sind, welche Farben und Formen und so weiter. Auch 3D-Effekte, Animationen und andere graphische Spielereien sollen nur im View und in keinem anderen Programmteil zu finden sein.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Sämtliche GUI-Komponenten werden selbstverständlich vollständig von der GUI verwaltet, also auch alle Buttons und andere Arten von Komponenten, die für Nutzereingaben da sind. Die ersten Schritte der Auswertung von Nutzereingaben sind noch im View, wir werden gleich sehen, wie die Nutzerwünsche dann Richtung Model und Controller weitergereicht werden.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller*: wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Der Controller enthält dann nur noch die übergeordneten Aspekte, zum Beispiel, ob gerade ein Spiel läuft oder nicht.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Schachprogramm (vereinfacht):

- ***Model***:
  - **In welcher Zeile / Spalte welche Figur steht.**
  - **Vorwissen über den Spieler.**
  - **Subroutine zur Berechnung des nächsten Zugs.**
- ***View***:
  - **Darstellung des Schachbretts am Bildschirm.**
  - **Annahme der Nutzereingaben.**
- ***Controller***: **wer spielt, ob gerade ein Spiel läuft, ein neues begonnen werden soll, wer wo im Ranking steht usw.**
  - **Möglicherweise eher im Model statt im Controller („Geschäftslogik“).**

Gerade zwischen Model und Controller sind die Übergänge, wie schon angedeutet, natürlich fließend. Wir hatten schon gesagt, dass die Geschäftslogik eigentlich in den Controller soll, aber oft genug ganz gut ins Modell hineinpassen würde. Daher kann man sich die Aspekte, die wir hier dem Controller zugerechnet haben, zumindest teilweise auch gut im Model vorstellen.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Wie kommunizieren diese drei Komponenten des Programms – Model, View und Controller – nun zweckmäßigerweise miteinander? Sie sehen auf dieser Folie, dass die notwendige Kommunikation zwischen den drei Komponenten eigentlich eher gering und gut eingrenzbar ist.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Die einzige Nutzerinteraktion, die beim Model ankommen muss, ist der jeweilige Spielzug. Was der Nutzer genau mit der Maus anstellen musste, um die Figur von einem Feld auf ein anderes zu schieben, ist für das Model egal, das kann alles im View verarbeitet und dann wieder vergessen werden. Die einzige Information an das Model besteht aus den zwei Feldern, von wo nach wo die Figur vom Nutzer per Maus gezogen wurde.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Es sollte Aufgabe der View sein zu prüfen, ob der Nutzer die Maus wirklich von einem Feld des Schachbretts zu einem anderen gezogen hat, denn das hat mit den Abmessungen der Darstellung des Brettes zu tun. Es sollte aber *nicht* Aufgabe der View, sondern des Model sein zu prüfen, ob der Nutzer korrekt gezogen hat, also ob auf dem Startfeld tatsächlich eine Figur der eigenen Farbe stand und ob diese Figur tatsächlich nach den Regeln des Schachspiels in der momentanen Spielstellung auf das Zielfeld gezogen werden konnte.

Falls der Zug nicht korrekt war, muss der Nutzer eine entsprechende Information und eine Wiederholungsmöglichkeit bekommen. Das ist Aufgabe der View, und dafür muss die View die Information vom Model erhalten, dass der Spielzug inkorrekt war. Ist der Zug des Spielers hingegen korrekt, ist der Gegenzug des Schachprogramms an die View mitzuteilen, damit diese die Darstellung des Schachbretts entsprechend aktualisiert.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Je nachdem, was im Controller noch an Geschäftslogik verortet ist, müssen entsprechende Informationen an den Controller gesandt werden. Wenn weiterhin im Controller gesteuert werden soll, ob ein Spiel läuft oder nicht, dann muss das Model eine Notiz an den Controller schicken, sobald es feststellt, dass das momentan laufende Spiel nach den Schachregeln beendet ist.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Und vom View enthält der Controller alle notwendigen Informationen, wenn der Nutzer in die Geschäftslogik eingegriffen hat, zum Beispiel, indem er einen Button zum Starten eines neuen Spiels anklickt.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Sollte der Controller seinerseits in seinem Verantwortungsbereich aktiv werden, dann muss er die beiden anderen Komponenten natürlich entsprechend informieren.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflüsse:

- **Von der *View* zum *Model*: welchen Zug der Spieler gemacht hat.**
- **Vom *Model* zur *View*: ob der Spielerzug korrekt war und welchen Zug das Programm daraufhin gemacht hat.**
- **Vom Model zum Controller: ob das Spiel zu Ende ist und mit welchem Ergebnis.**
- **Von der View zum Controller: wenn Button für neues Spiel, Programmende, aktuelle Rankinganzeige o.ä. gedrückt wurde.**
- **Vom Controller zu Model und View: wenn neues Spiel begonnen, aktuelles Ranking angezeigt o.ä. werden soll.**

Ansonsten sind Model, View und Controller völlig unabhängig voneinander!

Der entscheidende Punkt ist, dass die hier skizzierten Interaktionen zwischen den drei Komponenten auch die einzigen sind, ansonsten gibt es keine Verbindung zwischen den dreien.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- **Fenster enthält Spielbrett und weitere Buttons.**
  - **Gezeichnet von der View.**
- **An jeder GUI-Komponente hängen spezifische Listener.**
- **Datenfluss von den Listenern ...**
  - **... am Spielbrett: Züge des Spielers → an das Model.**
  - **... an Buttons für Spielende u.ä. → an den Controller.**
  - **... an Buttons für Änderung des Darstellungsstils → an die View.**

Mit den bisher schon kennengelernten Möglichkeiten von GUIs ist es nicht allzu schwer zu durchschauen, wie die Umsetzung in Java aussehen könnte.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- Fenster enthält Spielbrett und weitere Buttons.
  - Gezeichnet von der View.
- An jeder GUI-Komponente hängen spezifische Listener.
- Datenfluss von den Listenern ...
  - ... am Spielbrett: Züge des Spielers → an das Model.
  - ... an Buttons für Spielende u.ä. → an den Controller.
  - ... an Buttons für Änderung des Darstellungsstils → an die View.

Wie die GUI eines solchen Schachprogramms prinzipiell aussehen würde, bedarf wohl keiner Erläuterung.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- Fenster enthält Spielbrett und weitere Buttons.
  - Gezeichnet von der View.
- An jeder GUI-Komponente hängen spezifische Listener.
- Datenfluss von den Listenern ...
  - ... am Spielbrett: Züge des Spielers → an das Model.
  - ... an Buttons für Spielende u.ä. → an den Controller.
  - ... an Buttons für Änderung des Darstellungsstils → an die View.

Und wir hatten schon geklärt, dass alles, was direkt mit der GUI zu tun hat, Aufgabe der View ist.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- **Fenster enthält Spielbrett und weitere Buttons.**
  - **Gezeichnet von der View.**
- **An jeder GUI-Komponente hängen spezifische Listener.**
- **Datenfluss von den Listenern ...**
  - **... am Spielbrett: Züge des Spielers → an das Model.**
  - **... an Buttons für Spielende u.ä. → an den Controller.**
  - **... an Buttons für Änderung des Darstellungsstils → an die View.**

***Erinnerung*: In Kapitel 10 hatten wir an verschiedenen Stellen schon gesehen, wie GUI-Komponenten mit dem Rest des Programms auf einfache Weise kommunizieren können, nämlich über Listener.**

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- **Fenster enthält Spielbrett und weitere Buttons.**
  - **Gezeichnet von der View.**
- **An jeder GUI-Komponente hängen spezifische Listener.**
- **Datenfluss von den Listenern ...**
  - **... am Spielbrett: Züge des Spielers → an das Model.**
  - **... an Buttons für Spielende u.ä. → an den Controller.**
  - **... an Buttons für Änderung des Darstellungsstils → an die View.**

Man könnte zum Beispiel das ganze Spielbrett in einem Canvas realisieren und bei Mausaktionen im Spielbrett über die Koordinaten im MouseEvent erfragen, in welchem der 64 Felder die Maus im Moment der Aktion war. An dieses Canvas würde man dann einen Listener hängen. Alternativ könnte man auch jedes der 64 Felder etwa als eigenes Label realisieren, und dann hätte jedes dieser Label einen eigenen Listener. Das sind alles Details der View, die in den beiden anderen Komponenten nicht sichtbar sind.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- **Fenster enthält Spielbrett und weitere Buttons.**
  - **Gezeichnet von der View.**
- **An jeder GUI-Komponente hängen spezifische Listener.**
- **Datenfluss von den Listenern ...**
  - **... am Spielbrett: Züge des Spielers → an das Model.**
  - **... an Buttons für Spielende u.ä. → an den Controller.**
  - **... an Buttons für Änderung des Darstellungsstils → an die View.**

Nutzerinteraktionen, die das Gesamtprogramm steuern sollen, gehen analog an den Controller.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Mögliche Umsetzung in Java:

- Fenster enthält Spielbrett und weitere Buttons.
  - Gezeichnet von der View.
- An jeder GUI-Komponente hängen spezifische Listener.
- Datenfluss von den Listenern ...
  - ... am Spielbrett: Züge des Spielers → an das Model.
  - ... an Buttons für Spielende u.ä. → an den Controller.
  - ... an Buttons für Änderung des Darstellungsstils → an die View.

Die GUI könnte natürlich auch Möglichkeiten besitzen, mit denen der Nutzer das Look&Feel steuert, etwa die Farben oder die Größe der Darstellung. Solche Möglichkeiten fallen in den Aufgabenbereich der View, und die Nutzereingaben sollten daher an die View gehen. Das Model und der Controller sollten überhaupt nicht damit behelligt werden.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ChessBoardListener extends MouseAdapter {
   private ChessModel model;
   private ChessCoordinates coordinates;
   private boolean numberOfClicksOdd;
   public ChessBoardListener ( ChessModel model, ChessCoordinates coordinates ) {
      this.model = model;
      this.coordinates = coordinates;
      numberOfClicksOdd = false;
   }
   .....
}
```

Ein einzelnes Beispiel für eine geeignete Listener-Klasse soll hier genügen. Alle anderen Listener-Klassen sind analog. Der beispielhafte Listener auf dieser Folie soll Mausaktionen auf dem Spielbrett abhorchen, um daraus den Zug zu berechnen und diesen an das Model zu geben.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ChessBoardListener extends MouseAdapter {
   private ChessModel model;
   private ChessCoordinates coordinates;
   private boolean numberOfClicksOdd;
   public ChessBoardListener ( ChessModel model, ChessCoordinates coordinates ) {
      this.model = model;
      this.coordinates = coordinates;
      numberOfClicksOdd = false;
   }
   .....
}
```

***Erinnerung*: In Kapitel 10 hatten wir gesehen, dass diverse Listener-Interfaces jeweils von einer dazu passenden Adapter-Klasse implementiert werden, unter anderem MouseListener von MouseAdapter. Leitet man eine Listener-Klasse von MouseAdapter ab, dann muss man nur die Methoden implementieren, die man wirklich implementieren will, die anderen werden mit leerem Inhalt von MouseAdapter geerbt.**

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ChessBoardListener extends MouseAdapter {
  private ChessModel model;
  private ChessCoordinates coordinates;
  private boolean numberOfClicksOdd;
  public ChessBoardListener ( ChessModel model, ChessCoordinates coordinates ) {
    this.model = model;
    this.coordinates = coordinates;
    numberOfClicksOdd = false;
  }
  ..........
}
```

Um dem Model Informationen zukommen zu lassen, muss der Listener einen Verweis auf ein Objekt haben, das dem Listener geeigneten Zugang zum Model gewährt. Wir nehmen an, dass ChessModel der Name der Klasse ist, von der das Schachmodell ein Objekt ist.

***Erinnerung*: Das hatten wir in Kapitel 10 auch bei den damaligen Beispielen für Listener-Klassen gesehen: Ein Objekt der Listener-Klasse bekommt einen Verweis auf jedes Objekt, mit dem es interagieren soll.**

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ChessBoardListener extends MouseAdapter {
   private ChessModel model;
   private ChessCoordinates coordinates;
   private boolean numberOfClicksOdd;
   public ChessBoardListener ( ChessModel model, ChessCoordinates coordinates ) {
      this.model = model;
      this.coordinates = coordinates;
      numberOfClicksOdd = false;
   }
   .....
}
```

Gemäß KISS-Prinzip und insbesondere gemäß Separation of Concerns lagern wir die Information darüber, wo das Schachbrett auf dem Bildschirm platziert ist und wie groß es dargestellt ist, in eine eigene Klasse mit aussagekräftigem Namen aus.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ChessBoardListener extends MouseAdapter {
   private ChessModel model;
   private ChessCoordinates coordinates;
   private boolean numberOfClicksOdd;
   public ChessBoardListener ( ChessModel model, ChessCoordinates coordinates ) {
      this.model = model;
      this.coordinates = coordinates;
      numberOfClicksOdd = false;
   }
   .....
}
```

Wir nehmen der Einfachheit halber an, dass der Nutzer erst auf das Startfeld und dann auf das Zielfeld klicken soll, um einen Zug zu machen. Beide Klicks sollen durch dasselbe Listener-Objekt behandelt werden. Dafür muss das Listener-Objekt mitzählen, ob es der erste oder der zweite Klick war, am Besten in einer booleschen Variable, weil die nach jedem zweiten Klick sozusagen wieder auf Start steht.

***Nebenbemerkung*: Natürlich würde man bei einem realen Schachprogramm Drag&Drop anbieten, aber hier geht es ja um anderes, nämlich darum, wie der Informationsfluss vom View zum Model generell realisiert werden kann.**

***Erinnerung*: Drag&Drop hatten wir in Kapitel 10, speziell im Abschnitt zu Swing, ganz kurz erwähnt.**

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void mouseClicked ( MouseEvent event ) {
      int row = coordinates.getRow ( event.getX () );
      int column = coordinates.getColumn ( event.getY () );
      numberOfClicksOdd = ! numberOfClicksOdd;
      if ( numberOfClicksOdd )
         model.setStartField ( row, column );
      else {
         model.setTargetField ( row, column );
         model.checkUserMoveAndMakeOwnMove ();
      }
}
```

Wie gesagt, interessieren uns nur Klicks, keine anderen Operationen des Nutzers. Also implementieren wir nur die Methode mouseClicked der neuen Listener-Klasse ChessBoardListener.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void mouseClicked ( MouseEvent event ) {
    int row = coordinates.getRow ( event.getX () );
    int column = coordinates.getColumn ( event.getY () );
    numberOfClicksOdd = ! numberOfClicksOdd;
    if ( numberOfClicksOdd )
       model.setStartField ( row, column );
    else {
       model.setTargetField ( row, column );
       model.checkUserMoveAndMakeOwnMove ();
    }
}
```

In die Klasse ChessCoordinates war ja ausgelagert, wo das Schachbrett platziert ist und wie groß es ist. Damit ist Klasse ChessBoard die ideale Autorität, um Zeile und Spalte aus den Koordinaten des Mausklicks zu bestimmen. Daher würden für Klasse ChessBoard Methoden wie getRow und getColumn zu definieren sein.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void mouseClicked ( MouseEvent event ) {
    int row = coordinates.getRow ( event.getX () );
    int column = coordinates.getColumn ( event.getY () );
    numberOfClicksOdd = ! numberOfClicksOdd;
    if ( numberOfClicksOdd )
       model.setStartField ( row, column );
    else {
       model.setTargetField ( row, column );
       model.checkUserMoveAndMakeOwnMove ();
    }
}
```

Wie schon gesagt, soll dasselbe Objekt der neuen Listener-Klasse sowohl den Klick auf das Startfeld wie auch den Klick auf das Zielfeld empfangen und verarbeiten. Dafür muss Klasse ChessModel entsprechende Methoden bereitstellen, zum Beispiel so wie hier, je eine Methode, die Zeile und Spalte des Feldes als Parameter bekommen.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void mouseClicked ( MouseEvent event ) {
    int row = coordinates.getRow ( event.getX () );
    int column = coordinates.getColumn ( event.getY () );
    numberOfClicksOdd = ! numberOfClicksOdd;
    if ( numberOfClicksOdd )
       model.setStartField ( row, column );
    else {
       model.setTargetField ( row, column );
       model.checkUserMoveAndMakeOwnMove ();
    }
}
```

Da immer abwechselnd Startfeld und Zielfeld angeklickt werden, muss der entsprechende boolesche Zähler nach jedem Mausklick – also in jedem Aufruf von Methode mouseClicked – geändert werden.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void mouseClicked ( MouseEvent event ) {
    int row = coordinates.getRow ( event.getX () );
    int column = coordinates.getColumn ( event.getY () );
    numberOfClicksOdd = ! numberOfClicksOdd;
    if ( numberOfClicksOdd )
       model.setStartField ( row, column );
    else {
       model.setTargetField ( row, column );
       model.checkUserMoveAndMakeOwnMove ();
    }
}
```

Nachdem auch das Zielfeld gesetzt ist, hat Klasse ChessModel zu prüfen, ob der Zug nach den Schachregeln gültig ist, und ändert auf dieser Basis dann die innere Darstellung der aktuellen Spielsituation. Sollte der Zug *nicht* gültig sein, informiert das Model die View über einen anderen, hier nicht gezeigten Listener, so dass die View dem Nutzer eine Fehlermeldung ausgeben und eine Wiederholungsmöglichkeit anbieten kann. Ist hingegen der Zug des Nutzers gültig, wird die schon erwähnte Subroutine im Model zur Berechnung eines Gegenzugs gestartet.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void mouseClicked ( MouseEvent event ) {
    int row = coordinates.getRow ( event.getX () );
    int column = coordinates.getColumn ( event.getY () );
    numberOfClicksOdd = ! numberOfClicksOdd;
    if ( numberOfClicksOdd )
       model.setStartField ( row, column );
    else {
       model.setTargetField ( row, column );
       model.checkUserMoveAndMakeOwnMove ();
    }
}
```

Die Methode checkUserMoveAndMakeOwnMove ist natürlich wieder ein Beispiel für aussagekräftige Namen, die alles beinhalten, was die Methode so macht. Dass die Methode mit einem Listener die View benachrichtigt, falls der Zug inkorrekt war, muss nicht in den Namen der Methode hinein. Denn durch den Namensbestandteil checkUserMove ist jeder Programmierer, der mit dieser Methode umgeht, ausreichend gewarnt, dass die Prüfung des Nutzerzugs ja sicherlich noch irgendwelche Konsequenzen haben wird, und wird diese sicherlich dann in der Dokumentation der Methode nachschlagen.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Nach diesem ersten Beispiel sollten die Vorteile von MVC jetzt eigentlich schon evident sein, hier noch einmal zusammengefasst.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Wir haben ja gesehen, dass die Schnittstellen zwischen Model, View und Controller recht klein und einfach sind. Wenn in einer der Komponenten ein Fehler korrigiert werden muss oder eine der drei Komponenten weiterentwickelt werden soll, dann wäre der Idealfall, dass die Schnittstellen so bleiben können, wie sie vorher waren. In diesem Idealfall sind die anderen beiden Komponenten überhaupt nicht betroffen. Aber auch wenn die Schnittstellen anzupassen sind, kann man aller Erfahrung nach darauf hoffen, dass an den jeweils anderen beiden Komponenten meist doch nur einfache, überschaubare Anpassungen notwendig sein werden.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Wenn die Komponenten sich schon so wenig gegenseitig beeinflussen, dass man *eine* der Komponenten meist unabhängig von den anderen beiden Komponenten ändern kann, dann kann man auch noch einen Schritt weitergehen und gleich austauschen statt weiterzuentwickeln.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Das ist insbesondere wichtig, wenn man Software plattformunabhängig implementieren will. Beispielsweise ein Smartphone und ein Desktop benötigen in vielen Fällen unterschiedliche Views, auch wenn das dahinterstehende Model dasselbe ist.

***Nebenbemerkung*: Das Stichwort dazu heißt *responsive design*.**

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Die Plattform ist nicht die einzige Art von Kontext. Da kommt dann die Möglichkeit ins Spiel, auch den Controller auszutauschen.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Im nächsten, also dem zweiten und letzten Beispiel wird *dieser* Aspekt zum Tragen kommen: Wieso soll es nur *eine* View für ein Model geben, wieso nicht gleich *mehrere*?

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorteile von MVC:

- **Änderungen an einer der drei Komponenten ziehen minimale Änderungen an den anderen beiden Komponenten nach sich.**
- **View und Controller sind problemlos austauschbar.**
  - **Für eine neue Plattform oder ein neues Look&Feel ist nur der View auszutauschen.**
  - **Model und View lassen sich in einen neuen Kontext (=Controller) einbauen.**
- **Mehrere Views gleichzeitig sind möglich.**
  - **Verschieden gestaltet, auf verschiedenen Geräten usw.**
  - **Konsistent mit dem Model → auch konsistent untereinander.**

Der entscheidende Vorteil von MVC ist hier die Konsistenz: Jede einzelne View wird über entsprechende Listener immer mit dem Model konsistent gehalten, und damit sind auch die Views immer untereinander konsistent. Zwei verschiedene Nutzer mit zwei verschiedenen Views bekommen konsistente Informationen dargestellt, und die Eingriffe des *einen* Nutzers erscheinen dann auch zeitnah beim *anderen* Nutzer.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Das zweite Beispiel handeln wir nur ganz kurz ab. Es geht hier nicht mehr darum zu sehen, *wie* MVC funktioniert, das hatten wir ja schon beim ersten Beispiel gesehen. Hier geht es eher darum, einen Eindruck zu vermitteln, wie hilfreich MVC als Strukturierungsmittel für große Softwarepakete ist, und wie sich MVC in großen Softwarepaketen darstellt.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Es geht um Reisendeninformationssysteme wie das der Deutschen Bahn oder wie die Systeme der lokalen Verkehrsverbünde. Dies ist ein Forschungsthema der Arbeitsgruppe des Autors dieser Folien. Daraus sind schon etliche Themen für unser Algorithmenpraktikum und auch etliche Abschlussarbeiten entstanden – vielleicht auch Ihre in ein paar Semestern?

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Selbstverständlich lässt sich ein solches Thema nicht auf einer einzigen Folie abhandeln – auch der Begriff „stark vereinfacht“ ist immer noch eine Untertreibung.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Das Model enthält alles, was die Betriebssituation insgesamt ausmacht. Dazu gehören erst einmal statische Informationen wie der Fahrplan, aber auch aktuelle Informationen zu Verspätungen, Ausfällen und so weiter. Ein weiteres Beispiel für Daten im Model sind Kunden, die ihre geplante Fahrverbindung im System haben hinterlegen lassen, um im Falle einer Betriebsstörung per SMS oder Email informiert zu werden. Natürlich sind das nur wenige Beispiele für die Vielzahl von Informationen, die im Modell hinterlegt sind und verarbeitet werden.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Sie haben sicherlich schon selbst mit unterschiedlichen Views auf das zugrundeliegende Model zu tun gehabt.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Dies sind ein paar der Views, mit denen Sie als Normalkunde schon in Berührung gekommen sein könnten. Nicht nur das Look&Feel ist unterschiedlich, sondern auch die Auswahl der dargestellten Informationen und Ihre Interaktionsmöglichkeiten können sich von View zu View drastisch unterscheiden.

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model*: Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*):**
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller*: Computernetzwerk.**

Bedienstete des Bahnunternehmens haben natürlich teilweise ganz andere Views, in denen sie auch interne Informationen sehen, und die ihnen auch die Möglichkeit zum Eingreifen in den Betriebsablauf geben, zum Beispiel in der Fahrdienstleitung.

***Nebenbemerkung*: Besuchen Sie einmal das Eisenbahnbetriebsfeld der Deutschen Bahn, das ist in Darmstadt, gleich hinter dem Hauptbahnhof. Auch dort entstehen viele Abschlussarbeiten, auch mit Informatikthemen.**

MVC

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Reisendeninformationssystem (stark vereinfacht):

- ***Model***: **Fahrplandaten, aktuelle Betriebssituation, hinterlegte Trips .....**
- ***View*(*s*)**:
  - **Für Reisende: Anzeigetafeln, Fahrkartenautomaten, Internetportal, App .....**
  - **Für Bedienstete: .....**
- ***Controller***: **Computernetzwerk.**

Es gibt einen Aspekt in einem solchen Softwarepaket, der mit Sicherheit weder ins Model noch in die View gehört, nämlich die Organisation der Netzwerkverbindungen. Das ist definitiv eine Aufgabe des Controllers.

TECHNISCHE UNIVERSITÄT DARMSTADT

Konformität

Kommen wir zum letzten Aspekt in unserer Aufzählung von Ansätzen für fehlervermeidenden Entwurf: Konformität. Hier geht es um die Konformität von Subtypen zu ihren Basistypen, also rein objektorientiert, funktionale Abstraktion bleibt außen vor.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung Liskov Substitution Principle (LSP):

Jede Aussage über das logische Verhalten der Basisklasse muss auch für die abgeleitete Klasse gelten.

→Keine bösen Überraschungen, wenn statischer und dynamischer Typ nicht übereinstimmen:

```
public class X
  public void m ( ) { ..... }
}
public class Y extends X {
   public void m ( ) { ..... }
}
```

```
X a = new X ();
X b = new Y ();

a.m ();
b.m ();
```

Dreh- und Angelpunkt ist das LSP, das wir in Kapitel 12, Abschnitt zur Korrektheit von Klassen, kurz gestreift hatten; oben auf dieser Folie noch einmal in abstrakter Form formuliert.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung Liskov Substitution Principle (LSP):

Jede Aussage über das logische Verhalten der Basisklasse muss auch für die abgeleitete Klasse gelten.

→Keine bösen Überraschungen, wenn statischer und dynamischer Typ nicht übereinstimmen:

```
public class X
  public void m ( ) { ..... }
}
public class Y extends X {
  public void m ( ) { ..... }
}
```

```
X a = new X ();
X b = new Y ();

a.m ();
b.m ();
```

Das Prinzip ist in Java vor allem relevant beim Überschreiben von Methoden. Hier wieder einmal unser rein illustratives Standardbeispiel: Eine Methode wird in der Basisklasse definiert und implementiert und in der abgeleiteten Klasse überschrieben.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung Liskov Substitution Principle (LSP):

Jede Aussage über das logische Verhalten der Basisklasse muss auch für die abgeleitete Klasse gelten.

→Keine bösen Überraschungen, wenn statischer und dynamischer Typ nicht übereinstimmen:

```
public class X
   public void m ( ) { ..... }
}
public class Y extends X {
    public void m ( ) { ..... }
}
```

```
X a = new X ();
X b = new Y ();

a.m ();
b.m ();
```

Um diesen Punkt geht es: Wenn Methode m mit statischem Typ X aufgerufen wird, dann soll Methode m sich auch konform zu X verhalten, obwohl die Implementation von m in Y eine andere als in X ist.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

Was heißt das? Welche Möglichkeiten haben wir dann überhaupt noch bei der Implementation der überschreibenden Methode, wenn sie ja irgendwie gleich der überschriebenen Methode sein soll?

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

Nun, zunächst einmal gibt es Situationen, in denen eine Methode in der abgeleiteten Klasse sogar überschrieben werden *muss*, weil die Implementation der Methode in der Basisklasse nicht auf die Logik der abgeleiteten Klasse passt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

In Kapitel 08 hatten wir etliche Beispiele dazu gesehen, hier nur eines stellvertretend für die anderen.

Wir haben dort gesehen, dass die von Reader abgeleiteten Klassen wie Bauteile aus einem Baukasten kombinierbar sind, und jede dieser Klassen fügt bestimmte Funktionalität hinzu.

Zum Beispiel die Klasse BufferedReader fügt einen internen Puffer zur Zwischenspeicherung der eingelesenen Daten hinzu. Die Methode read muss dann aus dem Puffer statt aus der Datenquelle einlesen und muss daher in BufferedReader überschrieben werden.

Das ist auch unproblematisch: Nach außen hin ist der Puffer nicht sichtbar, die Methode read liefert dieselben Ergebnisse wie bei Klasse Reader, das LSP ist also erfüllt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

***Erinnerung*: In Kapitel 02 hatten wir dieses Beispiel ausführlich gesehen.**

Dieses Beispiel ist etwas anders gelagert, denn die Methode paint ist in der Basisklasse abstrakt, es müssen also keine zwei Implementationen der Methode konform zueinander sein. Trotzdem ist das LSP einzuhalten, denn die Methode paint hat auch schon in der Basisklasse eine Nachbedingung: Das graphische Objekt soll auf die Zeichenfläche gezeichnet werden, die hinter dem Parameter von paint steht, und zwar nach Maßgabe der Attribute in der Basisklasse, also Randfarbe, Füllfarbe, Drehwinkel und so weiter.

Diese Nachbedingung hatten wir bei der Implementation von paint für diverse abgeleitete Klassen eingehalten und somit das LSP erfüllt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

***Erinnerung*: Dieses Beispiel hatten wir in Kapitel 01f in einem eigenen Abschnitt gesehen, zweite eigene Roboterklasse.**

Dieses Beispiel ist etwas anders gelagert. Die Implementation der Methode move in der abgeleiteten Klasse hat ja ein etwas anderes Verhalten als in Klasse Robot selbst, weil sie die Programmausführung nach jedem Schritt kurz verzögert. Aber: Zeitvorgaben gehören *nicht* zur Spezifikation von move in Robot, und daher kann das LSP durch zeitliche Verzögerungen nicht verletzt werden.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

Über reine Anpassung an den Subtyp hinaus bietet das LSP einen großen Spielraum, um in Subtypen die Funktionalität zu ändern, nämlich erweiternd beziehungsweise verfeinernd. Auch dazu ein paar Beispiele.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

***Erinnerung*: Das war die erste eigene Roboterklasse in Kapitel 01f.**

Die abgeleitete Klasse hat eine weitere Funktionalität, die in der Basisklasse noch nicht definiert ist, nämlich Rechtsdrehen. Die in der Basisklasse Robot definierten Methoden werden nicht überschrieben, daher kann das LSP nicht verletzt werden.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

***Erinnerung*: Abschnitt zur Klasse Frame in Kapitel 10.**

Die Klasse Frame fügt zu einem Fenster noch einen Rahmen hinzu, was die Klasse Window selbst noch nicht hat. Aber solche Zusatzfunktionalität wird durch die Darstellungsinvariante der Klasse Window nicht ausgeschlossen, also ist das LSP nicht verletzt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

Ähnlich verhält es sich bei diesem Beispiel aus demselben Kapitel. In der Klasse Button wird alle Funktionalität für die Darstellung und die Funktion von Buttons erst hinzugefügt. Wird ein Button-Objekt über den statischen Typ JComponent angesprochen, verhält es sich auch gemäß der Darstellungsinvariante von JComponent, die die spezifische Funktionalität von Buttons selbstverständlich nicht ausschließt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

Dieses Beispiel ist vielleicht etwas diffiziler. Konzeptuell ist ein Container ja zunächst einmal etwas völlig anderes als eine Komponente: Ein Container *enthält* Komponenten. Aber hier wurde dieses intuitive Konzept so erweitert, dass man gesagt hat, auch ein Container ist eine bestimmte Art von Komponente, nämlich eine, die auch andere Komponenten enthalten kann.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Erlaubt **nach LSP beim Überschreiben im Subtyp sind:**

- **Anpassung der Funktionalität an die Besonderheiten des Subtyps**
  - **Beispiel: Methode read von Reader / BufferedReader**
  - **Beispiel: Methode paint von GeomShape2D / Circle**
  - **Beispiel: Robot und SlowMotionRobot**
- **Zusätzliche bzw. verfeinerte Funktionalität, die keinen Effekt auf die Stellen hat, an denen Verhalten des Basistyps erwartet wird**
  - **Beispiel: Robot und SymmTurner**
  - **Beispiel: Window und Frame**
  - **Beispiel: JComponent und JButton**
  - **Beispiel: JComponent und JContainer**
  - **Beispiel: DMatrix und DSquareMatrix**

***Erinnerung*: Abschnitt zur Korrektheit von Klassen in Kapitel 12.**

Dieses Beispiel ist jetzt wirklich etwas knifflig.

Dass das LSP erfüllt ist, liegt daran, dass die Klasse DMatrix für allgemeine Matrizen keine Methoden hat, die der Logik von quadratischen Matrizen widersprechen würden. Konkret hatten wir das festgemacht an dem Beispiel einer Methode, die die Zeilenzahl ändert und die Spaltenzahl festhält oder umgekehrt. Eine solche Methode widerspricht offensichtlich dem Konzept einer quadratischen Matrix. Wären solche Methoden in der Basisklasse, dann könnte keine noch so geniale abgeleitete Klasse für quadratische Matrizen das LSP einhalten, es geht einfach logisch nicht.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Verboten **nach LSP beim Überschreiben im Subtyp ist:**

- **Änderung der Funktionalität, so dass ein Effekt an den Stellen auftreten kann, an denen der Basistyp verwendet wird**
  - **Beispiel: Robot.move wird in abgeleiteter Klasse so überschrieben, dass Roboter gleich zwei Schritte vorwärtsgehen**
  - **Beispiel: DMatrix und DSquareMatrix mit Methoden zum voneinander unabhängigen Ändern von Zeilenzahl und Spaltenzahl**
  - **Beispiel: eine java.util.List implementierende Klasse zählt die Positionen ab 1 statt ab 0**

Nachdem wir geklärt haben, was das LSP so alles erlaubt, schauen wir uns jetzt noch anhand von ein paar Beispielen an, wie typischerweise Verletzungen des LSP aussehen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Verboten **nach LSP beim Überschreiben im Subtyp ist:**

- **Änderung der Funktionalität, so dass ein Effekt an den Stellen auftreten kann, an denen der Basistyp verwendet wird**
  - **Beispiel: Robot.move wird in abgeleiteter Klasse so überschrieben, dass Roboter gleich zwei Schritte vorwärtsgehen**
  - **Beispiel: DMatrix und DSquareMatrix mit Methoden zum voneinander unabhängigen Ändern von Zeilenzahl und Spaltenzahl**
  - **Beispiel: eine java.util.List implementierende Klasse zählt die Positionen ab 1 statt ab 0**

Das Problem ist immer dasselbe, wie schon pointiert vor ein paar Folien bei der Wiederholung des LSP formuliert: Das LSP ist verletzt, wenn Überraschungen auftreten können, sobald man ein Objekt eines Subtyps durch einen Basistyp anspricht.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Verboten **nach LSP beim Überschreiben im Subtyp ist:**

- **Änderung der Funktionalität, so dass ein Effekt an den Stellen auftreten kann, an denen der Basistyp verwendet wird**
  - **Beispiel: Robot.move wird in abgeleiteter Klasse so überschrieben, dass Roboter gleich zwei Schritte vorwärtsgehen**
  - **Beispiel: DMatrix und DSquareMatrix mit Methoden zum voneinander unabhängigen Ändern von Zeilenzahl und Spaltenzahl**
  - **Beispiel: eine java.util.List implementierende Klasse zählt die Positionen ab 1 statt ab 0**

Nehmen Sie an, eine Referenz von Klasse Robot verweist auf ein Objekt von irgendeiner abgeleiteten Klasse. Wenn der Aufruf von Methode move zu einem abweichenden Verhalten führt, zum Beispiel mehrere Schritte vorwärts statt nur einem, dann ist das schon eine unliebsame Überraschung. Stellen Sie sich beispielsweise den Murks vor, den ein solcher Roboter im Fallbeispiel in Kapitel 03c anrichten würde!

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Verboten **nach LSP beim Überschreiben im Subtyp ist:**

- **Änderung der Funktionalität, so dass ein Effekt an den Stellen auftreten kann, an denen der Basistyp verwendet wird**
  - **Beispiel: Robot.move wird in abgeleiteter Klasse so überschrieben, dass Roboter gleich zwei Schritte vorwärtsgehen**
  - **Beispiel: DMatrix und DSquareMatrix mit Methoden zum voneinander unabhängigen Ändern von Zeilenzahl und Spaltenzahl**
  - **Beispiel: eine java.util.List implementierende Klasse zählt die Positionen ab 1 statt ab 0**

Dieses Beispiel hatten wir in Kapitel 12: Die Klasse für allgemeine Matrizen bietet Methoden an, mit denen eine quadratische Matrix nichtquadratisch gemacht werden kann. Diese Methoden müssen auch in jeder abgeleiteten Klasse entweder ererbt oder überschrieben werden. Aber es ist schon rein logisch überhaupt keine Implementation dieser Methoden möglich, die einerseits gemäß LSP konform zur Implementation in der Basisklasse ist und andererseits der Logik einer quadratischen Matrix nicht widerspricht.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Verboten **nach LSP beim Überschreiben im Subtyp ist:**

- **Änderung der Funktionalität, so dass ein Effekt an den Stellen auftreten kann, an denen der Basistyp verwendet wird**
  - **Beispiel: Robot.move wird in abgeleiteter Klasse so überschrieben, dass Roboter gleich zwei Schritte vorwärtsgehen**
  - **Beispiel: DMatrix und DSquareMatrix mit Methoden zum voneinander unabhängigen Ändern von Zeilenzahl und Spaltenzahl**
  - **Beispiel: eine java.util.List implementierende Klasse zählt die Positionen ab 1 statt ab 0**

Noch ein weiteres sicherlich einsichtsreiches Beispiel, nun aus Kapitel 07, das nochmals etwas anders gelagert ist: Zur Darstellungsinvariante des Interface List gehört ja, dass die Positionen ab 0 gezählt werden. In jeder implementierenden Klasse müssen sich alle Methoden daran halten. Zum Beispiel muss die Methode add mit Index 3 das neue Element an der *vierten* Position einfügen, nicht an der *dritten*.

Nichteinhaltung solcher subtiler Feinheiten beim Ableiten von Klassen ist ein beliebter Programmierfehler.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterstützung für das LSP in Java:

- **Der Kopf der überschreibenden Methode darf nur nach engen Regeln vom Kopf der überschriebenen Methode abweichen.**
- **Konkret:**
  - **Zugriffsrechte private → ε → protected → public.**
  - **Wenn der Rückgabetyp der überschriebenen Methode ein Referenztyp ist, darf der Rückgabetyp der überschreibenden Methode ein Subtyp davon sein.**
  - **Die überschreibende Methode darf Subtypen derjenigen Exception-Klassen werfen, die die überschriebene Methode wirft.**

Programmiersprachen, die objektorientierte Konzepte enthalten, unterstützen das LSP in der Regel, indem sie bestimmte Verletzungen des LSP von vornherein ausschließen oder zumindest erschweren. Wir sehen uns das konkret bei Java an.

***Erinnerung*: Diese Folie ist nur eine Wiederholung aus Kapitel 03c, Abschnitt zur Signatur und zu Überschreiben und Überladen von Methoden.**

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterstützung für das LSP in Java:

- **Der Kopf der überschreibenden Methode darf nur nach engen Regeln vom Kopf der überschriebenen Methode abweichen.**
- **Konkret:**
  - **Zugriffsrechte private → ε → protected → public.**
  - **Wenn der Rückgabetyp der überschriebenen Methode ein Referenztyp ist, darf der Rückgabetyp der überschreibenden Methode ein Subtyp davon sein.**
  - **Die überschreibende Methode darf Subtypen derjenigen Exception-Klassen werfen, die die überschriebene Methode wirft.**

Durch diese uns schon bekannte Regel wird für Folgendes gesorgt: Wenn eine Methode oder ein Attribut des statischen Typs an einer Programmstelle irgendwo außerhalb der Definition des statischen und des dynamischen Typs verwendbar ist, dann ist diese Methode beziehungsweise dieses Attribut des dynamischen Typs ebenfalls an dieser Programmstelle verwendbar. Es kann also keine Überraschungen geben dadurch, dass beispielsweise eine Methode in der Basisklasse public und in der abgeleiteten Klasse private ist und diese private-Methode über eine Referenz der Basisklasse doch aufgerufen werden kann.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterstützung für das LSP in Java:

- **Der Kopf der überschreibenden Methode darf nur nach engen Regeln vom Kopf der überschriebenen Methode abweichen.**
- **Konkret:**
  - **Zugriffsrechte private → ε → protected → public.**
  - **Wenn der Rückgabetyp der überschriebenen Methode ein Referenztyp ist, darf der Rückgabetyp der überschreibenden Methode ein Subtyp davon sein.**
  - **Die überschreibende Methode darf Subtypen derjenigen Exception-Klassen werfen, die die überschriebene Methode wirft.**

Wir hatten schon gesehen, dass der Rückgabetyp problemlos durch einen Subtyp ersetzt werden kann. Wenn die Rückgabe gemäß ihrem Typ verwendet wird, darf dahinter ja auch ein Objekt eines Subtyps stehen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterstützung für das LSP in Java:

- **Der Kopf der überschreibenden Methode darf nur nach engen Regeln vom Kopf der überschriebenen Methode abweichen.**
- **Konkret:**
  - **Zugriffsrechte private → ε → protected → public.**
  - **Wenn der Rückgabetyp der überschriebenen Methode ein Referenztyp ist, darf der Rückgabetyp der überschreibenden Methode ein Subtyp davon sein.**
  - **Die überschreibende Methode darf Subtypen derjenigen Exception-Klassen werfen, die die überschriebene Methode wirft.**

Auch bei Exceptions ist es analog kein Problem, wenn das von der überschreibenden Methode geworfene Objekt von einer Subklasse einer Klasse ist, die in der throws-Klausel der überschriebenen Klasse deklariert ist.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem bei Arrays in Java :

```
public class X { ..... }
public class Y extends X { ..... }

X[ ] a = new Y [ 357 ];
X x = a [ 356 ];
a [ 356 ] = new X();
```

Leider gibt es in Java eine Lücke in der Typsicherheit, die man bewusst in Kauf genommen hat, um Subtypen von Arrays zuzulassen. Das sehen wir uns an diesem Beispiel an.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem bei Arrays in Java :

```
public class X { ..... }
public class Y extends X { ..... }

X[ ] a = new Y [ 357 ];
X x = a [ 356 ];
a [ 356 ] = new X();
```

***Erinnerung*: In Kapitel 03b, Abschnitt zu Subtypen und statischen und dynamischem Typ, hatten wir schon gesehen, dass einer Arrayvariable von einem Basistyp durchaus ein Arrayobjekt von einem Subtyp zugewiesen kann.**

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem bei Arrays in Java :

```
public class X { ..... }
public class Y extends X { ..... }

X[ ] a = new Y [ 357 ];
X x = a [ 356 ];
a [ 356 ] = new X();
```

Solange nur *lesend* auf die Komponenten des Arrays zugegriffen wird, ist auch alles gut: Einer Variablen vom Basistyp kann man natürlich eine Komponente eines Arrays vom Subtyp zuweisen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem bei Arrays in Java :

```
public class X { ..... }
public class Y extends X { ..... }

X[ ] a = new Y [ 357 ];
X x = a [ 356 ];
a [ 356 ] = new X();
```

Aber beim *Setzen* einer Arraykomponente sieht das leider anders aus. Die rot unterlegte Anweisung geht problemlos durch den Compiler. Aber würde sie wirklich ausgeführt, dann würde plötzlich eine Variable vom Subtyp – nämlich die Arraykomponente – auf ein Objekt vom Basistyp verweisen. Das darf nicht sein!

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem bei Arrays in Java :

```
public class X { ..... }
public class Y extends X { ..... }

X[ ] a = new Y [ 357 ];
X x = a [ 356 ];
a [ 356 ] = new X();        // ArrayStoreException
```

Daher wird in einem solchen Fall die Zuweisung nicht ausgeführt, sondern es wird eine Exception von einer eigens für diesen Fall definierten Exception-Klasse geworfen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem bei Arrays in Java :

```
public class X { ..... }
public class Y extends X { ..... }

X[ ] a = new Y [ 357 ];
X x = a [ 356 ];
a [ 356 ] = new X();        // ArrayStoreException
```

Achtung: abgeleitet von RuntimeException!

Damit nicht jeder schreibende Zugriff auf eine Arraykomponente durch einen try-catch-Block eingerahmt werden muss, ist das eine RuntimeException. Das heißt, wenn man Arraykomponenten setzen will, aber nicht genau weiß, woher das Arryayobjekt kommt, muss man sich in der Precondition-Klausel im Vertrag absichern, dass statischer und dynamischer Typ identisch sind. Alternativ muss man dann doch ArrayStoreException in irgendeiner Methode auf dem Call-Stack fangen, Oder man prüft erst einmal jedes

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
   .....
   public void scaleRadius1 ( double factor ) { ..... }
   public void scaleRadius2 ( double factor ) { ..... }
   .....
}

public class Circle extends Ellipse { ..... }
```

Als nächstes kommen wir zu einem klassischen Problem für die Konformität von Basistyp und Subtyp, dessen Name auf einem typischen Beispiel beruht. Genau dieses Beispiel sehen wir uns hier an.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
    .....
    public void scaleRadius1 ( double factor ) { ..... }
    public void scaleRadius2 ( double factor ) { ..... }
    .....
}

public class Circle extends Ellipse { ..... }
```

***Erinnerung*: das Fallbeispiel in Kapitel 02.**

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
  .....
  public void scaleRadius1 ( double factor ) { ..... }
  public void scaleRadius2 ( double factor ) { ..... }
  .....
}

public class Circle extends Ellipse { ..... }
```

Eine Klasse, die Ellipsen repräsentiert, sollte Möglichkeiten anbieten, um die beiden Radien unabhängig voneinander zu verändern.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
    .....
    public void scaleRadius1 ( double factor ) { ..... }
    public void scaleRadius2 ( double factor ) { ..... }
    .....
}

public class Circle extends Ellipse { ..... }
```

Jeder Kreis ist konzeptuell auch eine Ellipse. Daher würden wir gerne auch Kreise überall da verwenden können, wo mit Ellipsen gearbeitet wird. Es bietet sich also an, die Klasse für Kreise nicht direkt von GeomShape2D abzuleiten, sondern von der Klasse für Ellipsen.

Allerdings verletzen dann die beiden oben angedeuteten Methoden das LSP, denn natürlich müssen bei einem Kreis beide Radien immer identisch bleiben. Dieses Beispiel ist also ziemlich ähnlich zu unserem früheren Beispiel mit allgemeinen und quadratischen Matrizen, DMatrix und DSquareMatrix.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Circle extends GeomShape2D { ..... }

public class Ellipse extends Circle {
   .....
   public Ellipse ( double radius1, double radius2 ) { ..... }
   .....
}

Circle c = new Ellipse ( 2, 3 );
```

Es gibt auch gute Argumente dafür, umgekehrt die Klasse für Ellipsen von der Klasse für Kreise abzuleiten.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Circle extends GeomShape2D { ..... }

public class Ellipse extends Circle {
    .....
    public Ellipse ( double radius1, double radius2 ) { ..... }
    .....
}

Circle c = new Ellipse ( 2, 3 );
```

So sähe das dann aus.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Circle extends GeomShape2D { ..... }

public class Ellipse extends Circle {
  .....
  public Ellipse ( double radius1, double radius2 ) { ..... }
  .....
}

Circle c = new Ellipse ( 2, 3 );
```

Eine Klasse für Ellipsen sollte natürlich einen Konstruktor haben, der beide Radien mit beliebigen Werten initialisiert, die natürlich nicht identisch sein müssen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Circle extends GeomShape2D { ..... }

public class Ellipse extends Circle {
   .....
   public Ellipse ( double radius1, double radius2 ) { ..... }
   .....
}

Circle c = new Ellipse ( 2, 3 );
```

Und damit ist das LSP schon gleich bei der Einrichtung von Variable und Objekt verletzt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
   public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
   .....
}

public class Circle extends EllipseWithConstantRatio  {
   public  Circle ( double radius ) {
      super ( radius, radius );
   }
   .....
}
```

Auf dieser Folie sehen Sie einen eleganten Ausweg aus dem Kreis-Ellipse-Dilemma.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
    public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
    .....
}

public class Circle extends EllipseWithConstantRatio  {
    public  Circle ( double radius ) {
        super ( radius, radius );
    }
    .....
}
```

Die Grundidee hinter diesem Ansatz ist bei jedem Beispiel immer dasselbe: Wir definieren eine gemeinsame Basisklasse, die die gemeinsame Abstraktion der beiden im Konflikt stehenden Klassen darstellt.

Die gemeinsame Abstraktion von Kreis und Ellipse ist eine Ellipse, bei der zwar die Radien veränderlich, ihr *Verhältnis* zueinander aber unveränderlich ist.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
   public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
   .....
}

public class Circle extends EllipseWithConstantRatio  {
   public  Circle ( double radius ) {
      super ( radius, radius );
   }
   .....
}
```

Die beiden Radien werden im Konstruktor initialisiert. Damit ist das Verhältnis der beiden Radien ein für allemal festgelegt. Diese Basisklasse darf Methoden haben, die die Radien ändern, aber nur so, dass das Verhältnis der beiden Radien dabei gleich bleibt. Man kann sich beispielsweise eine Methode vorstellen, die den ersten Radius mit einem neuen, als Parameter gegebenen Wert überschreibt, und der zweite Radius wird automatisch mit überschrieben, so dass das Verhältnis der beiden gewahrt bleibt.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
   public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
   .....
}

public class Circle extends EllipseWithConstantRatio  {
   public  Circle ( double radius ) {
      super ( radius, radius );
   }
   .....
}
```

Ein Kreis ist in der Tat eine Ellipse mit einem konstanten Verhältnis der Radien, nämlich mit dem Verhältnis eins.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
   public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
   .....
}

public class Circle extends EllipseWithConstantRatio  {
   public  Circle ( double radius ) {
      super ( radius, radius );
   }
   .....
}
```

Daher hat der Konstruktor der Kreisklasse nur einen Parameter, und der Konstruktor der Basisklasse wird mit zweimal demselben Wert aufgerufen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
   public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
   .....
}

public class Ellipse extends EllipseWithConstantRatio  {
   public Ellipse ( double radius1, double radius2 ) {
      super ( radius1, radius2 );
   }
   double scaleRadius1  ( double factor ) { ..... }
   double scaleRadius2  ( double factor ) { ..... }
   .....
}
```

Auch die Klasse für Ellipsen kann man von der gemeinsamen Abstraktion ohne Verstoß gegen das LSP ableiten.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class EllipseWithConstantRatio extends GeomShape2D {
   public EllipseWithConstantRatio ( double radius1, double radius2 ) { ..... }
   .....
}

public class Ellipse extends EllipseWithConstantRatio  {
   public Ellipse ( double radius1, double radius2 ) {
      super ( radius1, radius2 );
   }
   double scaleRadius1  ( double factor ) { ..... }
   double scaleRadius2  ( double factor ) { ..... }
   .....
}
```

Jetzt ist es auch überhaupt kein Problem, Methoden wie etwa diese beiden in der Ellipsenklasse zu definieren, denn es wird ja keine Klasse mehr von der Ellipsenklasse abgeleitet, auf die diese Methoden nicht passen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
   .....
   public void scaleRadius1 ( double factor )
      throws UnsupportedOperationException { ..... }
   .....
}

public class Circle extends Ellipse { ..... }
```

Achtung: Ist eine RuntimeException!

Es gibt Situationen, in denen man das Dilemma nicht gut so auflösen kann wie auf der letzten Folie beschrieben. Für solche Fälle hat sich eine Konvention in Java entwickelt, wie man brachial dafür sorgen kann, dass das LSP trotzdem eingehalten wird. Das sehen Sie auf dieser Folie am selben Beispiel.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
    ..........
    public void scaleRadius1 ( double factor )
       throws UnsupportedOperationException { .......... }
    ..........
}

public class Circle extends Ellipse { .......... }
```

Achtung: Ist eine RuntimeException!

Was eben verworfen wurde, wird jetzt gemacht: Die Klasse für Kreise wird von der Klasse für Ellipsen abgeleitet.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
  ..........
  public void scaleRadius1 ( double factor )
     throws UnsupportedOperationException { .......... }
  ..........
}

public class Circle extends Ellipse { .......... }
```

Achtung: Ist eine RuntimeException!

Die Ellipsenklasse hat wieder Methoden, die für die Kreisklasse nicht sinnvoll sind.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
   .....
   public void scaleRadius1 ( double factor )
      throws UnsupportedOperationException { ..... }
   .....
}

public class Circle extends Ellipse { ..... }
```

Achtung: Ist eine RuntimeException!

Und dennoch leiten wir die Kreisklasse von der Ellipsenklasse ab.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
   ..........
   public void scaleRadius1 ( double factor )
      throws UnsupportedOperationException { .......... }
   ..........
}

public class Circle extends Ellipse { .......... }
```

Achtung: Ist eine RuntimeException!

Aber ein Detail ist jetzt anders: Die Methoden, die für abgeleiteten Klassen potentiell unpassend sind, werden schon in der Basisklasse so definiert, dass sie potentiell eine Exception werfen dürfen von einer Exception-Klasse, die extra für diesen Fall entwickelt worden ist. Die Implementation einer solchen Methode in der Ellipsenklasse wirft diese Exception nicht; die Implementation in der Kreisklasse tut nichts anderes als diese Exception zu werfen.

Man antizipiert also schon beim Design der Basisklasse, dass es mit abgeleiteten Klassen Probleme geben wird, und kommt dem durch diese throws-Klausel zuvor.

Das LSP ist damit formal erfüllt, aber wirklich vertrauenserweckend ist diese Vorgehensweise natürlich nicht, ...

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Kreis-Ellipse-Dilemma:

```
public class Ellipse extends GeomShape2D {
   .....
   public void scaleRadius1 ( double factor )
      throws UnsupportedOperationException { ..... }
   .....
}

public class Circle extends Ellipse { ..... }
```

Achtung: Ist eine RuntimeException!

... zumal diese Exception-Klasse nicht gefangen werden muss, bei Nichtbeachtung also zum Programmabbruch führen kann.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Collection {
  .........
  boolean add ( T element ) throws
                                UnsupportedOperationException,
                                NullPointerException,
                                ClassCastException,
                                IllegalArgumentException,
                                IllegalStateException {
    .........
```

Dieser Trick findet sich an verschiedenen Stellen in der Standardbibliothek. Einen dieser Fälle hatten wir schon im Kapitel zu Generics und Collections gesehen, uns aber nicht im Detail angeschaut: Methode add von Interface Collection.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Collection {
  .........
  boolean add ( T element ) throws
                              UnsupportedOperationException,
                              NullPointerException,
                              ClassCastException,
                              IllegalArgumentException,
                              IllegalStateException {
    .........
```

Es gibt Klassen, die Interface Collection implementieren, aber statisch sind in dem Sinne, dass ein Objekt einer solchen Klasse durch eine Menge von Elementen im Konstruktor initialisiert wird und danach nicht mehr verändert werden kann. Eine solche Klasse kann die Methode add und auch andere Methoden nicht sinnvoll implementieren. Die Implementationen dieser Methoden in einer solchen Klasse tun daher nichts anderes als eine Exception von dieser Exception-Klasse zu werfen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class Bird extends Animal {
   ..........
   abstract public int getMaxFlightHeight ();
   ..........
}
```

Problem:

- **Nicht alle Vögel können fliegen.**
  - **Bei denen ist maxFlightHeight sinnlos.**
  - **Trotzdem haben sie die Methode, wenn sie von Bird abgeleitet werden.**
- **Auch manche andere Tiere können fliegen.**
  - **Kein gemeinsamer Basistyp für die Behandlung von Flugtieren.**

Zum Abschluss des Abschnitts über Konformität eine Erinnerung an ein Fallbeispiel, das wir schon in Kapitel 03b im Abschnitt zu Interfaces gesehen hatten.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class Bird extends Animal {
    .........
    abstract public int getMaxFlightHeight ();
    .........
}
```

Problem:

- Nicht alle Vögel können fliegen.
  - Bei denen ist maxFlightHeight sinnlos.
  - Trotzdem haben sie die Methode, wenn sie von Bird abgeleitet werden.
- Auch manche andere Tiere können fliegen.
  - Kein gemeinsamer Basistyp für die Behandlung von Flugtieren.

Das Beispiel besteht darin, die Hierarchie der biologischen Arten, Gattungen, Familien und so weiter in einer Vererbungshierarchie abzubilden. Das hier anzusprechende Problem lässt sich speziell mit Flugtieren lebensnah illustrieren.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class Bird extends Animal {
    .........
    abstract public int getMaxFlightHeight ();
    .........
}
```

Problem:

- Nicht alle Vögel können fliegen.
  - Bei denen ist maxFlightHeight sinnlos.
  - Trotzdem haben sie die Methode, wenn sie von Bird abgeleitet werden.
- Auch manche andere Tiere können fliegen.
  - Kein gemeinsamer Basistyp für die Behandlung von Flugtieren.

Vögel können fliegen, das lernt man schon im Kleinkindalter. Daher liegt es nahe, der Klasse für Vögel Methoden mitzugeben, die sich auf das Fliegen beziehen.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class Bird extends Animal {
   ..........
   abstract public int getMaxFlightHeight ();
   ..........
}
```

Problem:

- **Nicht alle Vögel können fliegen.**
  - **Bei denen ist maxFlightHeight sinnlos.**
  - **Trotzdem haben sie die Methode, wenn sie von Bird abgeleitet werden.**
- **Auch manche andere Tiere können fliegen.**
  - **Kein gemeinsamer Basistyp für die Behandlung von Flugtieren.**

Wir haben wieder das Problem wie beim Kreis-Ellipse-Dilemma, dass Methoden zum Fliegen nicht für alle potentiellen Subtypen von Klasse Bird sinnvoll sind.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class Bird extends Animal {
   .........
   abstract public int getMaxFlightHeight ();
   .........
}
```

Problem:

- Nicht alle Vögel können fliegen.
  - Bei denen ist maxFlightHeight sinnlos.
  - Trotzdem haben sie die Methode, wenn sie von Bird abgeleitet werden.
- Auch manche andere Tiere können fliegen.
  - Kein gemeinsamer Basistyp für die Behandlung von Flugtieren.

Und es gibt auch ein umgekehrtes Problem: Wenn man Funktionalität für Flugtiere schreibt, dann möchte man diese natürlich gerne nur einmal schreiben und dann auf *alle* Flugtiere gleichermaßen anwenden können. Man möchte nicht diese Funktionalität einmal für flugfähige Vögel, einmal für Fledermäuse und so weiter schreiben müssen. Man braucht also einen gemeinsamen Basistyp für Flugtiere, auf dem die Funktionalität für Flugtiere aufsetzen kann.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Möglicher Lösungsansatz:

```
public interface Flying {
   int getMaxFlightHeight ();
}

public class Bird extends Animal { ..... }

public class PasserineBird extends Bird implements Flying { ..... }

public class Chiropter extends Mammal implements Flying { ..... }
```

Dieses Problem kann man ganz gut so lösen, wie es auf dieser Folie vorgestellt wird.

***Nebenbemerkung*: In manchen anderen Programmiersprachen wird dieser Lösungsansatz durch spezielle Sprachkonstrukte unterstützt, das Stichwort für eigene Recherchen ist *Mixin*.**

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Möglicher Lösungsansatz:

```
public interface Flying {
    int getMaxFlightHeight ();
}

public class Bird extends Animal { ..... }

public class PasserineBird extends Bird implements Flying { ..... }

public class Chiropter extends Mammal implements Flying { ..... }
```

Das Fliegen wird in ein Interface ausgelagert. Es kommt dann dadurch wieder hinein, dass Klassen für Flugtiere dieses Interface implementieren, das ist der Mixin-Vorgang.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Möglicher Lösungsansatz:

```
public interface Flying {
   int getMaxFlightHeight ();
}

public class Bird extends Animal { ..... }

public class PasserineBird extends Bird implements Flying { ..... }

public class Chiropter extends Mammal implements Flying { ..... }
```

Die Klasse Bird hat dann nichts mehr mit Fliegen zu tun.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Möglicher Lösungsansatz:

```
public interface Flying {
   int getMaxFlightHeight ();
}

public class Bird extends Animal { ..... }
public class PasserineBird extends Bird implements Flying { ..... }
public class Chiropter extends Mammal implements Flying { ..... }
```

Wir suchen dann nach möglichst hoch in der biologischen Hierarchie angesiedelten Klassen, bei denen wir uns sicher sein können, dass alle Arten darunter fliegen können. „Möglichst hoch“ heißt: lieber eine Gattung als eine einzelne Art, lieber eine Familie als eine Gattung und so weiter.

Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Möglicher Lösungsansatz:

```
public interface Flying {
   int getMaxFlightHeight ();
}

public class Bird extends Animal { ..... }

public class PasserineBird extends Bird implements Flying { ..... }

public class Chiropter extends Mammal implements Flying { ..... }
```

Bei diesem Ansatz ist es vollkommen egal, ob es sich um einen Subtyp von Klasse Bird handelt oder nicht. Völlig analog kann man daher auch den Fledertieren sozusagen das Fliegen beibringen.

Damit ist der Abschnitt zu Konformität und das ganze Kapitel beendet.