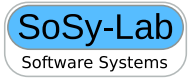
Formale Spezifikation und Verifikation

Wintersemester 2024

Prof. Dr. Gidon Ernst
gidon.ernst@lmu.de

Software and Computational Systems Lab
Ludwig-Maximilians-Universität München, Germany

November 29, 2024



Floyd/Hoare Logik

Floyd/Hoare Logik

Hoare-Logik: Prozeduren

Erinnerung: Hoares Ansatz

- *Hoare-Tripel* $\{P\}\ code\ \{Q\}$:
  Wenn vor einer Ausführung von Programmstück $code$ die Formel P gilt, dann gilt danach garantiert Q

- Regeln zur Konstruktion von *gültigen* Hoare-Tripeln für alle Programmkonstrukte aus der Grammatik
  (hier für eine abstrakte Programmiersprache)

$$\begin{aligned} \text{Anweisung } c ::= & \ \mathbf{skip} \ \mid \ \mathrm{x} = e \ \mid \ c_1; c_2 \\ \mid & \ \mathbf{if}\ \phi\ \mathbf{then}\ c_1\ (\mathbf{else}\ c_2)^? \\ \mid & \ \mathbf{while}\ \phi\ \mathbf{do}\ c \end{aligned}$$

Erinnerung: Hoares Ansatz

- *Hoare-Tripel* $\{P\}\ code\ \{Q\}$:
  Wenn vor einer Ausführung von Programmstück $code$ die Formel P gilt, dann gilt danach garantiert Q

- Regeln zur Konstruktion von *gültigen* Hoare-Tripeln für alle Programmkonstrukte aus der Grammatik
  (hier für eine abstrakte Programmiersprache)

$$\begin{aligned} \text{Anweisung } c ::= {} & \mathbf{skip} \mid \mathrm{x} = e \mid c_1; c_2 \\ \mid {} & \mathbf{if}\ \phi\ \mathbf{then}\ c_1\ (\mathbf{else}\ c_2)^? \\ \mid {} & \mathbf{while}\ \phi\ \mathbf{do}\ c \\ \mid {} & \mathrm{x} = \textsc{Proc}(e_1, \ldots, e_n) \qquad \text{Prozeduraufruf} \end{aligned}$$

- Ziel: Implementierung von einer Prozedur $\textsc{Proc}$ nur *einmal* verifizieren, aber möglicherweise an vielen Programmstellen *aufrufen*

Deklaration und Spezifikation von Prozeduren

Prozedurdeklaration

procedure $\textsc{Proc}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$
  returns y
  requires P
  ensures Q
begin
  code
end

Deklaration und Spezifikation von Prozeduren

Prozedurdeklaration

procedure PROC($x_1, \ldots, x_n$)
  returns y
  requires P
  ensures Q
begin
  code
end

Beispiel

procedure MAX(s)
  returns m
  requires s set with $s \neq \varnothing$
  ensures $m \in s \wedge \forall\, x \in s.\ m \geq x$
begin
  ...
end

Signatur von PROC

- Parameter $x_1, \ldots, x_n$
- Rückgabe-Variable y wird in *code* zugewiesen (statt **return**)

Wir spezifizieren einen *Kontrakt* für PROC

- Vorbedinung P: Formel über $x_1, \ldots, x_n$
- Nachbedinung Q: Formel über Rückgabe y und $x_1, \ldots, x_n$ (Startwerte)

Analog zu Dafny, aber hier nur eine Vor/Nachbedingung, nur eine Rückgabevariable, keine **return** Anweisung, Typisierung ignoriert

Hoare-Regel für Prozeduraufruf

Prozedurdeklaration

procedure PROC($\mathsf{x}_1, \ldots, \mathsf{x}_n$)
  returns y
  requires P
  ensures Q
begin
  code
end

Hoare-Regel für Prozeduraufruf

Prozedurdeklaration

procedure $\textsc{Proc}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$
  returns y
  requires P
  ensures Q
begin
  code
end

Beweisregel Prozeduraufruf

$$\frac{\phi \Longrightarrow P' \qquad Q' \Longrightarrow \psi}{\{\phi\}\ \mathsf{z} = \textsc{Proc}(e_1, \ldots, e_n)\ \{\phi \wedge \psi\}}\ \textsc{Call}$$

Einsetzen der Argumente e_i für Parameter x_i
$P' \equiv P[\mathsf{x}_1 \mapsto e_1, \ldots, \mathsf{x}_n \mapsto e_n]$

. . . sowie Ersetzten der Rückgabe y in Q durch die Zugewiesene Variable z
$Q' \equiv Q[\mathsf{x}_1 \mapsto e_1, \ldots, \mathsf{x}_n \mapsto e_n, \mathsf{y} \mapsto \mathsf{z}]$

Hoare-Regel für Prozeduraufruf

Prozedurdeklaration

procedure $\textsc{Proc}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$
  returns y
  requires P
  ensures Q
begin
  code
end

Beweisregel Prozeduraufruf

$$\frac{\phi \Longrightarrow P' \qquad Q' \Longrightarrow \psi}{\{\phi\}\ \mathsf{z} = \textsc{Proc}(e_1, \ldots, e_n)\ \{\phi \wedge \psi\}}\ \textsc{Call}$$

Einsetzen der Argumente e_i für Parameter x_i
$P' \equiv P[\mathsf{x}_1 \mapsto e_1, \ldots, \mathsf{x}_n \mapsto e_n]$

. . . sowie Ersetzten der Rückgabe y in Q durch die Zugewiesene Variable z
$Q' \equiv Q[\mathsf{x}_1 \mapsto e_1, \ldots, \mathsf{x}_n \mapsto e_n, \mathsf{y} \mapsto \mathsf{z}]$

Bedingung: z kommt in e_i und ϕ nicht vor (sonst dort umbenennen in neue Variable)

Bedingung: $\textsc{Proc}$ hat keine Seiteneffekte

Beispiel: Prozeduraufruf

Prozedurdeklaration

procedure MAX(s)
  returns m
  requires $\underbrace{\mathsf{s} \text{ set with } \mathsf{s} \neq \varnothing}_{P}$
  ensures $\underbrace{\mathsf{m} \in s \wedge \forall\, x \in s.\, \mathsf{m} \geq x}_{Q}$

Beweisregel Prozeduraufruf für MAX

$$\frac{\phi \Longrightarrow P' \qquad Q' \Longrightarrow \psi}{\{\phi\}\ \mathsf{z} = \textsc{Max}(e)\ \{\phi \wedge \psi\}}\ \textsc{Call}$$

$P' \equiv P[\mathsf{s} \mapsto e]$

$Q' \equiv Q[\mathsf{s} \mapsto e, \mathsf{m} \mapsto \mathsf{z}]$

Beispiel: Prozeduraufruf

Prozedurdeklaration

procedure MAX(s)
  returns m
  requires $\underbrace{\mathsf{s} \text{ set with } \mathsf{s} \neq \varnothing}_{P}$
  ensures $\underbrace{\mathsf{m} \in s \wedge \forall\, x \in s.\ \mathsf{m} \geq x}_{Q}$

Beweisregel Prozeduraufruf für MAX

$$\frac{\phi \Longrightarrow P' \qquad Q' \Longrightarrow \psi}{\{\phi\}\ \mathsf{z} = \textsc{Max}(e)\ \{\phi \wedge \psi\}}\ \textsc{Call}$$

$P' \equiv P[\mathsf{s} \mapsto e]$

$Q' \equiv Q[\mathsf{s} \mapsto e, \mathsf{m} \mapsto \mathsf{z}]$

Anwendung der Regel Ist das Tripel korrekt?

$$\{\underbrace{\mathsf{s}_1 = \{1\} \cup s_0}_{\phi}\}\ \mathsf{k} = \textsc{Max}(\mathsf{s}_1)\ \{\phi \wedge \underbrace{\mathsf{k} \geq 1}_{\psi}\}$$

- Regel*instanz* wobei wir z, e als k, s_1 wählen
- $P' \equiv P[\mathsf{s} \mapsto \mathsf{s}_1] \equiv (\mathsf{s}_1 \neq \varnothing)$
- $Q' \equiv Q[\mathsf{s} \mapsto \mathsf{s}_1, \mathsf{m} \mapsto \mathsf{k}] \equiv (\mathsf{k} \in \mathsf{s}_1 \wedge \forall\, x \in \mathsf{s}_1.\ \mathsf{k} \geq x)$

Beispiel: Prozeduraufruf (Nebenrechnungen)

- ϕ

Beispiel: Prozeduraufruf (Nebenrechnungen)

- $\phi \equiv (\mathsf{s}_1 = \{1\} \cup s_0)$

Beispiel: Prozeduraufruf (Nebenrechnungen)

- $\phi \equiv (\mathsf{s}_1 = \{1\} \cup s_0)$ also $\mathsf{s}_1 \neq \varnothing$

Beispiel: Prozeduraufruf (Nebenrechnungen)

- $\phi \equiv (\mathsf{s}_1 = \{1\} \cup s_0)$ also $\mathsf{s}_1 \neq \varnothing \equiv P'$ ✓
- Q'

Beispiel: Prozeduraufruf (Nebenrechnungen)

- $\phi \equiv (\mathsf{s}_1 = \{1\} \cup s_0)$ also $\mathsf{s}_1 \neq \varnothing \equiv P'$ ✓
- $Q' \equiv (\mathsf{k} \in \mathsf{s}_1 \wedge \forall\, x \in \mathsf{s}_1.\ \mathsf{k} \geq x)$

Beispiel: Prozeduraufruf (Nebenrechnungen)

- $\phi \equiv (\mathsf{s}_1 = \{1\} \cup s_0)$ also $\mathsf{s}_1 \neq \varnothing \equiv P'$ ✓
- $Q' \equiv (\mathsf{k} \in \mathsf{s}_1 \wedge \forall\, x \in \mathsf{s}_1.\ \mathsf{k} \geq x)$
  insbesondere für $x = 1$ (da $1 \in \mathsf{s}_1$) gilt $\mathsf{k} \geq 1$

Beispiel: Prozeduraufruf (Nebenrechnungen)

- $\phi \equiv (\mathsf{s}_1 = \{1\} \cup s_0)$ also $\mathsf{s}_1 \neq \varnothing \equiv P'$ ✓
- $Q' \equiv (\mathsf{k} \in \mathsf{s}_1 \wedge \forall\, x \in \mathsf{s}_1.\ \mathsf{k} \geq x)$
  insbesondere für $x = 1$ (da $1 \in \mathsf{s}_1$) gilt $\mathsf{k} \geq 1 \equiv \psi$ ✓

Verifikation von Deklarationen

Prozedurdeklaration

procedure PROC($\mathsf{x}_1, \ldots, \mathsf{x}_n$)
  returns y
  requires P
  ensures Q
begin
  code
end

Verifikation von Deklarationen

Prozedurdeklaration

procedure $\textsc{Proc}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$
  returns y
  requires P
  ensures Q
begin
  code
end

Beweisabriss (Schema)

Annahme für die Implementierung

- Vorbedingung gilt
- Startwerte x_i^{pre} der Parameter

$$\{P \wedge x_0^{\mathsf{pre}} = \mathsf{x}_0 \wedge \cdots \wedge x_n^{\mathsf{pre}} = \mathsf{x}_n\}$$
$$\vdots$$
$$code$$
$$\vdots$$
$$\{Q[\mathsf{x}_0 \mapsto x_0^{\mathsf{pre}}, \ldots, \mathsf{x}_n \mapsto x_n^{\mathsf{pre}}]\}$$

Zu zeigen ist die Nachbedingung für

- *finalen* Wert y der Rückgabevariable
- *initiale* Werte x_i^{pre} der Parameter

Bemerkung: Dafny verbietet lokale Zuweisungen auf Parameter, Java erlaubt es

Diskussion: Modularität

Modul—*eine funktional geschlossene Einheit [die] einen bestimmten Dienst bereitstellt*
(Gabler Lexikon)

Hier

- Modul: Prozedur
- Dienst: Berechnung der Ausgabe für gegebene Eingabe

Diskussion: Modularität

Modul—*eine funktional geschlossene Einheit [die] einen bestimmten Dienst bereitstellt*
(Gabler Lexikon)

Hier

- Modul: Prozedur
- Dienst: Berechnung der Ausgabe für gegebene Eingabe

Kontrakt = Vereinbarung zwischen Aufrufer und Implementierung

- Aufrufer muss Vorbedingung etablieren, darf Nachbedingung annehmen
- Prozedur darf Vorbedingung annehmen, muss Nachbedingung etablieren

Diskussion: Modularität

Modul—*eine funktional geschlossene Einheit [die] einen bestimmten Dienst bereitstellt*
(Gabler Lexikon)

Hier

- Modul: Prozedur
- Dienst: Berechnung der Ausgabe für gegebene Eingabe

Kontrakt = Vereinbarung zwischen Aufrufer und Implementierung

- Aufrufer muss Vorbedingung etablieren, darf Nachbedingung annehmen
- Prozedur darf Vorbedingung annehmen, muss Nachbedingung etablieren
- ✓ abstrakte Beschreibung der Prozedur möglich
- ✓ Entkopplung der Verifikation (Kontrakt ist eine Schnittstelle)
- ✓ bei Austausch der Implementierung: Beweis des Aufrufers bleibt gültig (sofern sich der Kontrakt nicht ändert)
- ✗ bei einfache Prozeduren: Code $\approx$ Spezifikation

Was Sie wissen und können sollten

- Über welche Variablen sind die Vor-/Nachbedingung einer Prozedur definiert und auf welche "Zeitpunkte" der Ausführung verweisen diese?
- Wie sieht die Regel zum Prozeduraufruf aus?
- Wie ersetzt man Prozedurparameter durch die Argumente an der Aufrufstelle?
- Wie wird die Implementierung einer Prozedur verifiziert?
- In wie fern ist der heute gezeigte Ansatz “modular”?

- Zum Nachdenken: Warum ist es *sinnvoll* dass die Parametervariablen in der Nachbedingung auf die Startwerte (vom Aufruf) verweisen?
  Spezifizieren sie dazu beispielhaft:
  procedure INC1(`i`) **returns** `j` **begin** `j = i+1` **end**
  procedure INC2(`i`) **returns** `j` **begin** `i = i+1; j = i` **end**

Verifikation von Objektorientierten Programmen

Agenda

- Spezifikation von Klassen
  - Externe *logische* Datensicht (mit Mengen, Sequenzen, ...)
  - Kontrakte für Methoden
- Verifikation von Klassen
  - Klasseninvarianten
  - Vererbungsprinzip

Praxisbezug

Einige Tools und Sprachen für objektorientierte Verifikation

- Dafny
- KeY (Java, embedded Java)
- VeriFast (Java)
- VerCors (Java)
- Java Modeling Language/OpenJML (Java)
- SPARK (Ada)
- EiffelStudio
- Spec#

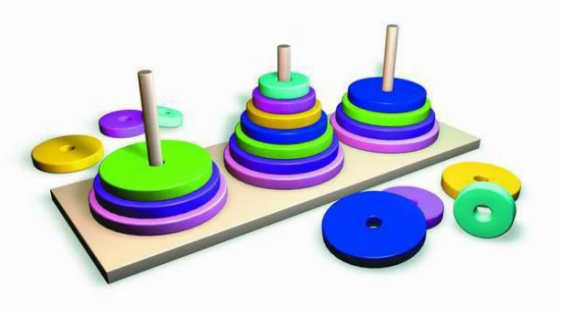
Unterschiedliche Ansätze, Anwendungsbereiche, Zielsetzungen, ...

Praxisbezug (mit KeY, Karlsruhe Institute of Technology, 2015)

Fehler in Standardsortieralgorithmus mit formalen Methoden aufgedeckt

Android, Java und Groovy nutzen alle den TimSort-Algorithmus. Informatiker eines Verbundprojekts konnten mit Hilfe eines von ihnen entwickelten Tools nun einen Fehler in der Implementierung feststellen und beheben.

Lesezeit: **1 Min.** In Pocket speichern 115



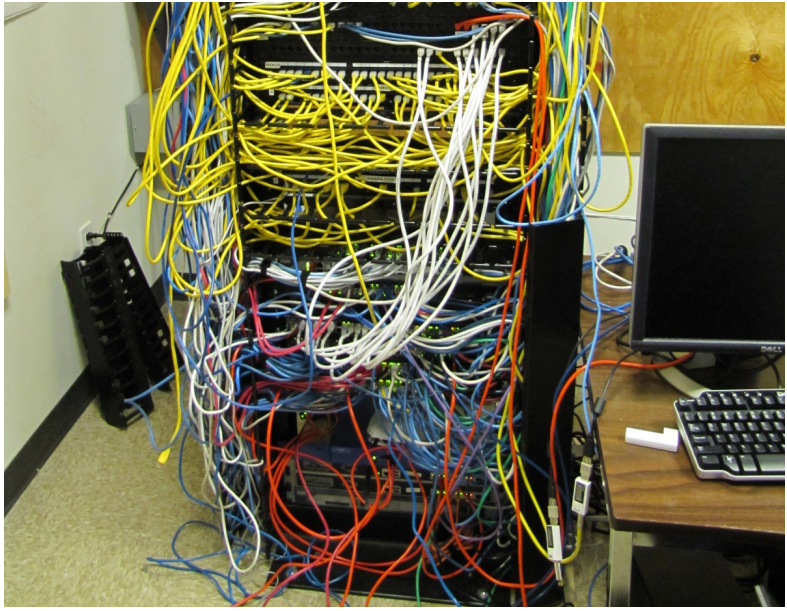
10.03.2015 08:17 Uhr | Developer

Von Julia Schmidt

Verifikation von Objektorientierten Programmen

Spezifikation von Interfaces und Klassen

Objektorientierte Programmierung



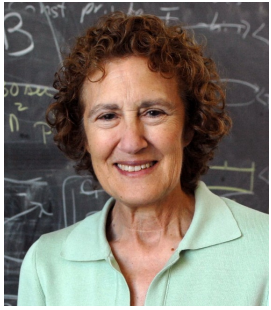
Objektorientierte Programmierung

Ziel: “Ordnung in das Chaos”

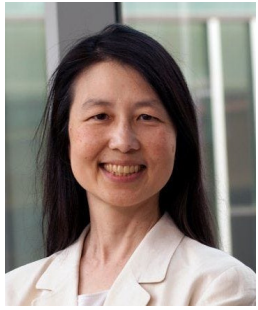
Strukturierung der Spezifikation und Verifikation entlang objektorientierter Prinzipien

- *Abstraktion* an Schnittstellen
- *Kapselung* interner Zustände/Berechnungen
- *Vererbung* als Verfeinerung in der Spezifikations und Modellhierarchie (vgl. Einführungsfolien)

Historisches...



Barbara Liskov

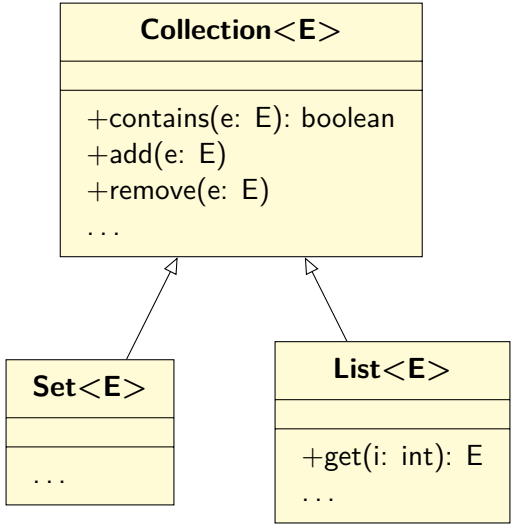


Jeanette Wing

A behavioral notion of subtyping (1994)

Spezifikations- und Beweisprinzipien der objektorientierte Programmierung

Beispiel einer Hierarchie



Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

`boolean contains(E e)`	"Returns true if this collection contains the specified element"
`boolean add(E e)`	"Ensures that this collection contains the specified element"
`boolean remove(E e)`	"Removes [one] instance the specified element from this collection"
`int size()`	"Returns the number of elements in this collection"
`boolean equals(Object o)`	"Compares the specified object with this collection for equality"

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

`boolean contains(E e)`	"Returns true if this collection contains the specified element"
`boolean add(E e)`	"Ensures that this collection contains the specified element"
`boolean remove(E e)`	"Removes [one] instance the specified element from this collection"
`int size()`	"Returns the number of elements in this collection"
`boolean equals(Object o)`	"Compares the specified object with this collection for equality"

- Explizite Garantien beziehen Operationen aufeinander, z.B.
  `c.add(e); assert c.contains(e);`

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

`boolean contains(E e)`	"Returns true if this collection contains the specified element"
`boolean add(E e)`	"Ensures that this collection contains the specified element"
`boolean remove(E e)`	"Removes [one] instance the specified element from this collection"
`int size()`	"Returns the number of elements in this collection"
`boolean equals(Object o)`	"Compares the specified object with this collection for equality"

- Explizite Garantien beziehen Operationen aufeinander, z.B.
  `c.add(e); assert c.contains(e);`
- Werden Duplikate gezählt?

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

`boolean contains(E e)`	"Returns true if this collection contains the specified element"
`boolean add(E e)`	"Ensures that this collection contains the specified element"
`boolean remove(E e)`	"Removes [one] instance the specified element from this collection"
`int size()`	"Returns the number of elements in this collection"
`boolean equals(Object o)`	"Compares the specified object with this collection for equality"

- Explizite Garantien beziehen Operationen aufeinander, z.B.
  `c.add(e); assert c.contains(e);`
- Werden Duplikate gezählt? Es kommt darauf an™ (`List` vs `Set`)
  `int n = c.size(); c.add(e); assert n+1 == c.size();`

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

`boolean contains(E e)`	"Returns true if this collection contains the specified element"
`boolean add(E e)`	"Ensures that this collection contains the specified element"
`boolean remove(E e)`	"Removes [one] instance the specified element from this collection"
`int size()`	"Returns the number of elements in this collection"
`boolean equals(Object o)`	"Compares the specified object with this collection for equality"

- Explizite Garantien beziehen Operationen aufeinander, z.B.
  `c.add(e); assert c.contains(e);`
- Werden Duplikate gezählt? Es kommt darauf an™ (`List` vs `Set`)
  `int n = c.size(); c.add(e); assert n+1 == c.size();`
- Wie funktioniert "Gleichheit"?

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

"A collection represents a group of [...] elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. [...]"

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html

`boolean contains(E e)`	"Returns true if this collection contains the specified element"
`boolean add(E e)`	"Ensures that this collection contains the specified element"
`boolean remove(E e)`	"Removes [one] instance the specified element from this collection"
`int size()`	"Returns the number of elements in this collection"
`boolean equals(Object o)`	"Compares the specified object with this collection for equality"

- Explizite Garantien beziehen Operationen aufeinander, z.B.
  `c.add(e); assert c.contains(e);`
- Werden Duplikate gezählt? Es kommt darauf an™ (`List` vs `Set`)
  `int n = c.size(); c.add(e); assert n+1 == c.size();`
- Wie funktioniert "Gleichheit"? Es kommt wieder darauf an™
  `if(c1.equals(c2)) { c1.add(e); c1.remove(e); assert c1.equals(c2); }`

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

```
public interface Set<E> extends Collection<E>
```

```
public interface List<E> extends Collection<E>
```

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

```
public interface Set<E> extends Collection<E>
```

"A collection that contains no duplicate elements [...] The `Set` interface places additional stipulations [...] on the contracts of the `add`, `equals` and `hashCode` methods"

`boolean add(E e)` "Adds the specified element to this set if it is not already present"

```
public interface List<E> extends Collection<E>
```

Stipulation: Bedingung, Vereinbarung, Vorgabe

Spezifikation von Klassen: javadoc

```
public interface Collection<E> extends Iterable<E>
```

```
public interface Set<E> extends Collection<E>
```

"A collection that contains no duplicate elements [...] The `Set` interface places additional stipulations [...] on the contracts of the `add`, `equals` and `hashCode` methods"

`boolean add(E e)` "Adds the specified element to this set if it is not already present"

```
public interface List<E> extends Collection<E>
```

"Unlike sets, lists typically allow duplicate elements. More formally, [...]"

`boolean add(E e)` "Appends the specified element to the end of this list"

Stipulation: Bedingung, Vereinbarung, Vorgabe

Welche Klasse? (I)

```
Collection<Integer> c = new ???();

c.add(1);
c.add(1);
c.remove(1);
c.contains(1); // false
```

Welche Klasse? (I)

```
Collection<Integer> c = new ???();

c.add(1);
c.add(1);
c.remove(1);
c.contains(1); // false
```

Set

- Keine Duplikate—2. Aufruf von add tut nichts

Welche Klasse? (II)

```
Collection<Integer> c = new ???();

c.add(1);
c.add(1);
c.remove(1);
c.contains(1); // true
```

Welche Klasse? (II)

```
Collection<Integer> c = new ???();

c.add(1);
c.add(1);
c.remove(1);
c.contains(1); // true
```

List

- Duplikate werden beibehalten

Welche Klasse? (III)

```
Collection<Integer> c1 = new ???();
Collection<Integer> c2 = new ???();

c1.add(1); c1.add(1); c1.add(2); c1.remove(1);
c2.add(2); c2.add(1);
c1.equals(c2); // true
```

Welche Klasse? (III)

```
Collection<Integer> c1 = new ???();
Collection<Integer> c2 = new ???();

c1.add(1); c1.add(1); c1.add(2); c1.remove(1);
c2.add(2); c2.add(1);
c1.equals(c2); // true
```

Gegencheck mit *mathematischen Datentypen*

Welche Klasse? (III)

```
Collection<Integer> c1 = new ???();
Collection<Integer> c2 = new ???();

c1.add(1); c1.add(1); c1.add(2); c1.remove(1);
c2.add(2); c2.add(1);
c1.equals(c2); // true
```

Gegencheck mit *mathematischen Datentypen*

✗ Mengen: $\texttt{c1} = \{1, 1, 2\} \setminus \{1\} = \{2\}$ $\texttt{c2} = \{2, 1\}$

Welche Klasse? (III)

```
Collection<Integer> c1 = new ???();
Collection<Integer> c2 = new ???();

c1.add(1); c1.add(1); c1.add(2); c1.remove(1);
c2.add(2); c2.add(1);
c1.equals(c2); // true
```

Gegencheck mit *mathematischen Datentypen*

- ✗ Mengen: $\texttt{c1} = \{1, 1, 2\} \setminus \{1\} = \{2\}$ $\texttt{c2} = \{2, 1\}$
- ✗ Sequenzen: $\texttt{c1} = \langle \not{1}, 1, 2 \rangle$ oder $\texttt{c1} = \langle 1, \not{1}, 2 \rangle$ $\texttt{c2} = \langle 2, 1 \rangle$

Welche Klasse? (III)

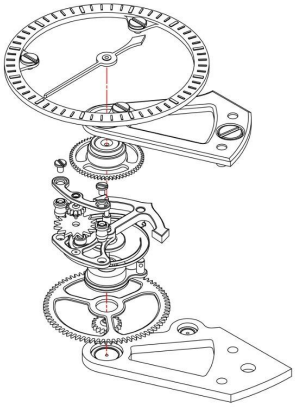
```
Collection<Integer> c1 = new ???();
Collection<Integer> c2 = new ???();

c1.add(1); c1.add(1); c1.add(2); c1.remove(1);
c2.add(2); c2.add(1);
c1.equals(c2); // true
```

Gegencheck mit *mathematischen Datentypen*

- ✗ Mengen: $\texttt{c1} = \{1, 1, 2\} \setminus \{1\} = \{2\}$ $\texttt{c2} = \{2, 1\}$
- ✗ Sequenzen: $\texttt{c1} = \langle 1, 1, 2\rangle$ oder $\texttt{c1} = \langle 1, 1, 2\rangle$ $\texttt{c2} = \langle 2, 1\rangle$
- ✓ Multimengen: $\texttt{c1} = \{\!| 1, 1, 2 |\!\}$ $\texttt{c2} = \{\!| 2, 1 |\!\}$
  - Duplikate werden beibehalten
  - Reihenfolge ist irrelevant

Qualitätssicherung (Technik)

Technisches System



Gesetze der Physik

Abstraktes Modell

verständlich, präzise

$$\frac{d \text{ hour}}{d \text{ minute}} = \frac{1}{60}$$

Objektorientierte Spezifikation [Liskov/Wing, 1994]

bag = **type**

uses BBag (bag **for** B)
for all b: bag

put = **proc** (i: int)
requires $| b_{pre}.elems | < b_{pre}.bound$
modifies b
ensures $b_{post}.elems = b_{pre}.elems \cup \{i\} \ \wedge \ b_{post}.bound = b_{pre}.bound$

get = **proc** ( ) **returns** (int)
requires $b_{pre}.elems \neq \{\}$
modifies b
ensures $b_{post}.elems = b_{pre}.elems - \{result\} \wedge result \in b_{pre}.elems \wedge$
$b_{post}.bound = b_{pre}.bound$

$card$ = **proc** ( ) **returns** (int)
ensures $result = | b_{pre}.elems |$

$equal$ = **proc** (a: bag) **returns** (bool)
ensures $result = (a = b)$

end bag

Fig. 1. A Type Specification for Bags

Spezifikation von Interfaces/Klassen

Zentrale Idee: definiere *abstraktes Modell* des Verhaltens

- Logische Sicht auf internen Zustand (Mengen, Sequenzen, ...)
- Spezifiziere *Außensicht* aller Methoden im Hinblick auf dieses Modell
  $\rightarrow$ Eingabe/Ausgabe Relationen über alten und neuen Zustand

Spezifikation von Interfaces/Klassen

Zentrale Idee: definiere *abstraktes Modell* des Verhaltens

- Logische Sicht auf internen Zustand (Mengen, Sequenzen, ...)
- Spezifiziere *Außensicht* aller Methoden im Hinblick auf dieses Modell
  $\rightarrow$ Eingabe/Ausgabe Relationen über alten und neuen Zustand

Beispiel: `List<E>`

- logische Zustandrepräsentation: Sequenz xs
- Konstruktor: Ergebniszustand $xs = \langle\,\rangle$ ist die leere Sequenz
- `add(e)`: $xs = \mathbf{old}(xs) \circ \langle e \rangle$
- `remove(e)`: $\exists 0 \leq i \leq |xs|.\ \mathbf{old}(xs) = xs[..i] \circ \langle e \rangle \circ xs[i..]$
  (nur sinnvoll falls e überhaupt vorhanden)

Spezifikation von Interfaces/Klassen

Zentrale Idee: definiere *abstraktes Modell* des Verhaltens

- Logische Sicht auf internen Zustand (Mengen, Sequenzen, ...)
- Spezifiziere *Außensicht* aller Methoden im Hinblick auf dieses Modell
  $\rightarrow$ Eingabe/Ausgabe Relationen über alten und neuen Zustand

Beispiel: `List<E>`

- logische Zustandrepräsentation: Sequenz xs
- Konstruktor: Ergebniszustand $xs = \langle\,\rangle$ ist die leere Sequenz
- `add(e)`: $xs = \mathbf{old}(xs) \circ \langle e \rangle$
- `remove(e)`: $\exists 0 \leq i \leq |xs|.\ \mathbf{old}(xs) = xs[..i] \circ \langle e \rangle \circ xs[i..]$
  (nur sinnvoll falls e überhaupt vorhanden)

Notwendig: mathematische Operationen auf Sequenzen

- Länge $|xs|$, Konkatenation $\circ$, Subsequenz bis/von Index $xs[..i]$ und $xs[i..]$
- Axiome/Theoreme, z.B. $|xs \circ ys| = |xs| + |ys|$
  $i \leq |xs| \Rightarrow xs[..i] \circ xs[i..] = xs$

Beispiel: Spezifikation von `List` (Auszug)

class LIST

model xs: sequence

method `add`(e)
ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

method `remove`(e)
requires $\exists 0 \leq i < |xs|.\ xs[i] = e$
ensures $\exists 0 \leq i \leq |xs|.\ \mathbf{old}(xs) = xs[..i] \circ \langle e \rangle \circ xs[i..]$

Beachte: dies entspricht nicht genau der textuellen Beschreibung von `List` in Java (dort hat `remove` keine Vorbedingung). Varianten können je nach Anwendungsfall und Anforderungen unterschiedlich ausfallen.

Beispiel: Spezifikation von `Set` (Auszug)

class SET

model s: (mathematical) set

method `add`(e)
ensures $s = \mathbf{old}(s) \cup \{e\}$

method `remove`(e)
requires $e \in s$
ensures $s = \mathbf{old}(s) \setminus \{e\}$

Auch hier sind wieder Varianten möglich.

Zusammenfassung und Diskussion

Die Spezifikation von Interfaces und Klassen besteht aus

- logischer Repräsentation interner Zustände
- Kontrakte aller Methoden bezüglich Zustandsänderungen

Die Spezifikation ist *unabhängig* von der konkreten Implementierung

- ✓ bezüglich der Algorithmen: analog zu Prozeduren
- ✓ Abstraktion von den intern verwendeten Datenstrukturen (z.B. Array, verkettete Liste)
- ✓ Wiederverwendung von Spezifikationen durch Vererbung:

```
public class HashSet<E> implements Set<E>
```

Ausblick:

- Verifikation von Implementierungen von Klassen bezüglich Spezifikationen
- Regeln für korrekte Vererbung (z.B. `Collection` vs `Set`/`List`)

License