Kapselung und Datenstruktur Liste

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

1 Kapselung und Strukturieren von Klassen

Objekte stellen eine bestimmte Funktionalität bzw. Dienste zur Verfügung.

- Aus *Anwendersicht* bedeutet dies, dass Objekte Nachrichten empfangen und Ergebnisse liefern können.
- Aus *Implementierungssicht* müssen Zustände und Funktionalität durch
  → objektlokale Attribute
  → Referenzen auf andere Objekte
  → Implementierung von Methoden
  realisiert werden.

Die ***Anwendungsschnittstelle*** eines Objekts besteht aus

- den Nachrichten, die es für Anwender zur Verfügung stellt;
- den Attributen, die für den direkten Zugriff von Anwendern bestimmt sind.

Das Ziel eines objekt-orientierten Designs ist es die Anwendungsschnittstelle genau festzulegen. Die Festlegung von Anwendungsschnittstellen ist eine Entwurfsentscheidung, d.h. sie wird bereits beim Entwurf festgelegt. Gleichzeitig soll der Zugriff "von außen" auf Implementierungsteile, die nur für internen Gebrauch bestimmt sind, verhindert werden. Direkter Zugriff auf Attribute beispielsweise muss nicht gewährt werden, sondern kann mit Nachrichten/Methoden realisiert werden (siehe Abschnitt "Zugriffsmethoden").

1.1 Information Hiding

Das Prinzip des ***Information Hiding*** (deutsch meist ***Geheimnisprinzip***) besagt, dass Anwendern nur die Informationen zur Verfügung stehen sollen, die zur Anwendungsschnittstelle gehören, und alle anderen Informationen und Implementierungsdetails für ihn verborgen und möglichst nicht zugreifbar sind.

Die Gründe für Information Hiding sind vielfältig: Zum einen vermeidet man die unsachgemäße Verwendung von Attributen. Es vereinfacht die Struktur von Software durch Reduktion der Abhängigkeiten zwischen ihren Teilen. Außerdem ermöglicht es den Austausch von Implementierungsteilen, ohne dass die Verwendung von Objekten beeinträchtigt wird.

Beispiele Die Darstellung von Adressen ändert sich regelmäßig und passt sich dabei den aktuellen Gegebenheiten an.

- **Postleitzahlen**: In (West-)Deutschland wurden bis 1962 zweistellige, danach vierstellige Postleitzahlen, seit 1993 schließlich fünfstellige Postleitzahlen verwendet. Es ist semantisch nicht sinnvoll mit Postleitzahlen zu rechnen. In vielen Ländern bestehen die Postleitzahlen aus bis zu 10 Zeichen und enthalten nicht nur Ziffern, sondern auch Buchstaben und Bindestriche.
- **IP-Adressen**: Das Internetprotokoll (IP) ist eines der wichtigsten Protokolle des Internets. Es legt fest, wie Geräte eindeutig addressiert werden können. Die Version IPv4 verwendet 32-Bit-Adressen und erlaubt es daher, Netze von bis zu 4.3 Milliarden direkt addressierbaren Geräten zu verwalten. Die Nachfolgeversion IPv6 nutzt 128-Bit-Adressen und erlaubt die direkte Adressierung von etwa $3,4 \cdot 10^{38}$ Geräten.

Programme, die von der tatsächliche Darstellung von Postleitzahlen bzw. IP-Adressen in Klassen abstrahieren und die Implementierungsdetails geheim halten, können deren Implementierung auf einfache Art ändern und an neue Anforderungen anpassen, indem die sie lediglich die String-Repräsentierung der Objekte nach außen sichtbar und verfügbar machen.

Information Hiding in Java Java ermöglicht es für Programmelemente ***Zugriffsbereiche*** zu deklarieren. Vereinfachend gesagt kann ein Programmelement nur innerhalb seines Zugriffsbereichs verwendet werden.
Java unterscheidet vier Arten von Zugriffsbereichen:

- nur innerhalb der eigenen Klasse (Modifikator `private`)
- nur innerhalb des eigenen Pakets[1] (ohne Modifikator)
- nur innerhalb des eigenen Pakets[1] und in Subklassen (Modifikator `protected`)
- ohne Zugriffsbeschränkung (Modifikator `public`)

Generell können Klassen, Attribute, Methoden und Konstruktoren mit diesen Zugriffsmodifikatoren deklariert werden.

[1] Pakete fassen Klassen zu einer größeren Einheit zusammen. Wir werden uns später noch näher mit Paketen beschäftigen.

Beispiel Um zu vermeiden, dass Programmierer die Klasse `RunnersDiary` falsch verwenden, in dem sie z.B. das Objektgeflecht der Liste kaputtmachen, kann man das Attribut `first` als privates Attribute deklarieren.

```
public class RunnersDiary {
    private String name;
    private Node first;
    public RunnersDiary(String name) {
        this.name = name;
    }
    // Liefert den Namen des Laeufers
    public String getName() {...}

    // Fuegt einen neuen Eintrag hinzu
    public void add(Entry e) {...}

    // Entfernt alle Eintraege zu einem Datum
    public void remove(Date d) {...}

    // Liefert das Element an Position index
    public Entry get(int index) {...}

    // Darstellung als String
    public String toString() {...}
}
```

Weitere Änderungen an der Implementierung von `RunnersDiary` sind nicht erforderlich. Die neue Schnittstelle erlaubt nun keinen direkten Zugriff mehr auf die Knoten der `Entry`-Liste.

- Es gibt keine Methode `getFirst` o.ä., und keine der Methoden liefert eine Referenz auf ein `Node`-Objekt.
- Außerhalb der Klasse `RunnersDiary` kann daher weder das Attribut `first` noch die Listenstruktur verändert werden.

Damit wird vermieden, dass Programmierer beispielsweise Zyklen in den Listen verursacht. Es erlaubt ausserdem die Implementierung der `Node`-Klasse beliebig abzuändern, ohne dass der Code, der Objekte der Klasse `RunnersDiary` verwendet, abgeändert werden muss.
Wir können beispielsweise die Klasse um ein Attribut `size` erweitern, das die Anzahl der Listenelemente enthält. Diese wird mit `0` initialisiert und muss mit jedem Einfügen eines neuen Elements inkrementiert werden.

```
    private String name;
    private int size;
    private Node first;

    RunnersDiary(String name) {
        this.name = name;
        this.size = 0;
        this.first = null;
    }

    // Liefert die Anzahl der Eintraege
    String getSize() {
```

```
      return this.size;
    }

    // Fuegt einen neuen Eintrag hinzu
    void add(Entry e) {
      ...
      this.size++;
    }
    ...
}
```

Die Schnittstelle erlaubt keinen direkten Zugriff auf die Länge der `Entry`-Liste.

- Außerhalb der Klasse `RunnersDiary` kann nur mittels `getSize()` die Anzahl der Listeneinträge abgefragt werden.
- Verwender der Klasse können das `size`-Attribut nicht beliebig verändern.

Damit wird vermieden, dass es Inkonsistenzen zwischen der Listenstruktur und dem Attribut gibt.

1.2 Zugriffsfunktionen

Der Zugriff auf Attribute wird häufig über spezifische *getter- / setter-* Methoden realisiert. Hier die beiden Varianten im Vergleich:

```
class AttributMitDirektemZugriff {
   public int attr;
}

class AttributZugriffUeberMethoden {
   private int attr;
   public int getAttr() {
    return attr;
   }
   public void setAttr(int a) {
    if (a > 0) {
      attr = a;
    }
   }
}
```

Die Verwendung von getter-/setter-Methoden erlaubt es insbesondere Modifikationen zu kontrollieren, beispielsweise

- das Überprüfen der Gültigkeit von neuen Attributwerten
- das Benachrichtigen von anderen Objekten bei Änderungen
  (⇒ Vorlesung zu “Beobachtermuster”)

Man kann so ausserdem erzwingen, dass Attributwerte nicht direkt verändert werden können, in dem z.B. keine setter-Methode zur Verfügung gestellt wird.

Frage 1:

1. Schreiben Sie die Klasse `Date` so um, dass die Felder nicht direkt zugreifbar sind.

```
class Date {
    int day;
    int month;
    int year;

    Date(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
}
```

2. Fügen Sie nun Getter-Methoden für die einzelnen Attribute hinzu.
3. Wie kann man die Erzeugung von ungültigen Datumswerten (z.B. 31.02.2018) verhindern?

Beispiel: Web-Seiten Die Klasse `W3Seite` implementiert eine einfache Repräsentierung von Web-Seiten mit Titelzeile und Inhalt.

```
   private String titel;
   private String inhalt;

   public W3Seite(String t, String i) {
      titel  = t;
      inhalt = i;
   }
   public String getTitel() {
    return titel;
   }
   public String getInhalt(){
    return inhalt;
   }
}
```

Die obige Klasse kann ersetzt werden durch die folgende Implementierung, ohne dass Anwender der Klasse davon betroffen werden. Diese alternative Klassendeklaration verwendet nur ein Attribut; der Titel wird dem Inhalt vorangestellt und mit dem `TITLE`-Tag markiert.

```
  private String seite;
  public W3Seite(String t, String i) {
    seite =  "<TITLE>" + t + "</TITLE>" + i ;
  }
  public String getTitel() {
    int ix = seite.indexOf("</TITLE>") - 7;
    return new String(seite.toCharArray(), 7, ix);
  }
```

```
  public String getInhalt() {
    int ix = seite.indexOf("</TITLE>") + 8;
    return new String(seite.toCharArray(), ix,
                      seite.length() - ix );
  }
}
```

Wie an den Beispielen erläutert, erlaubt Information Hiding konsistente Namensänderungen in versteckten Implementierungsteilen und das Verändern versteckter Implementierungsteile, soweit sie keine Auswirkungen auf die öffentliche Funktionalität haben.
Diese Auswirkungen können mitunter subtil, aber dennoch wichtig sein: Die zweite Implementierung von W3Seite kann z.B. nur dann anstelle der ersten benutzt werden, wenn Titel den Substring `</TITLE>` nicht enthalten.
Es gilt für einen guten Programmierstil in Java die Regel:

> Attribute sollten privat sein und nur in Ausnahmefällen öffentlich.

1.3 Grenzen der Zugriffskontrolle

Die Verwendung von Zugriffsmodifikatoren schließt in Java nicht automatisch aus, dass interne Eigenschaften von außen verändert werden können.

Frage 2: Die Verwendung von `private` führt **nicht** automatisch zur Kapselung.

```
public class A {
   private int[] werte;

   public void setWerte(int[] ar){
      for (int i = 0; i < ar.length; i++) {
        if (ar[i] < 0) {
          // Repariere Eintrag
          ar[i] = 0;
        }
      }
      this.werte = ar;
   }
}
```

In obigem Beispiel ist nicht sichergestellt, dass das Array `werte` keine negative Einträge enthält.
Geben Sie eine Beispielverwendung an, die eine negative Zahl in das `werte`-Array einträgt, ohne die Implementierung der Klasse `A` zu verändern.

2 Datenstruktur Liste

Wir haben in der Fallstudie zum Lauftagebuch Listen als eine erste klassische Datenstruktur kennengelernt. In diesem Abschnitt werden wir weitere Implementierungen von Listen kennenlernen und diese vergleichen.
Wir beginnen dazu mit einer formale Definition:

Definition Eine ***Liste über einem Typ*** T ist eine total geordnete Multimenge mit Elementen aus T (bzw. eine Folge, d.h. eine Abbildung $\mathbb{N} \to T$).
Eine Liste heißt ***endlich***, wenn sie nur endlich viele Elemente enthält.

Es gibt viele verschiedene Möglichkeiten Listen in Java zu implementieren. Wir betrachten hier zunächst drei Implementierungsvarianten:

1. als einfachverkettete Liste
2. als Array-Liste
3. als doppeltverkettete Liste

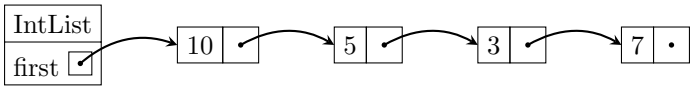
Dabei konzentrieren uns auf das Einfügen von neuen Elementen.
Der Einfachheit halber, betrachten wir hier zunächst nur Listen aus `int`-Werten. Um Listen abstrakt zu repräsentieren, verwenden wir hier die folgende Schreibweise: `[6,-3,84]` Die leere Liste wird dabei als `[]` repräsentiert.

2.1 Einfachverkettete Listen

Bei einfachverketteten Listen wird für jedes Listenelement ein Objekt mit zwei Attributen angelegt:

- zum Speichern des Elements
- zum Speichern der Referenz auf den Rest der Liste.

Die Liste mit den Elementen `[10,5,3,7]` erhält also folgende Repräsentation:



Die Listenknoten können dabei folgendermaßen implementiert werden (vgl. die Implementierung des Lauftagebuchs!):

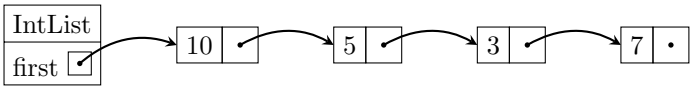
```
class Node {
    private int value;
    private Node next;

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
    int getValue() {
      return value;
    }
    Node getNext() {
      return next;
    }
    void setNext(Node n) {
      next = n;
    }

}
```

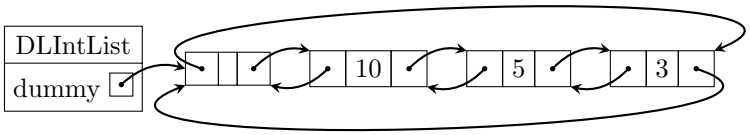
Die Listenknoten können nun zu einer Liste verkettet werden:

```
public class IntList {
  private Node first;

  IntList() {
    first = null;
  }

  // fuegt ein Element am Ende der Liste ein
  void add (int element) {
    Node newNode = new Node(element, null);
    if (first == null) {
      first = newNode;
    } else {
      Node n = first;
      Node nnext = n.getNext();
      while (nnext != null) {
        n = nnext;
      }
      n.setNext(newNode);
    }
  }
  // etc.
}
```

2.2 Doppeltverkettete Listen

Einfachverkettete Listen kann man nur effizient in eine Richtung durchlaufen, nämlich vom ersten Element zum letzten. Um die Liste auch "rückwärts" einfach zu durchlaufen, fügen wir eine weitere, doppelte Verkettung ein: Jeder Knoten zeigt nicht nur auf den Nachfolgerknoten (`Node`-Attribute `next`), sondern auch auf den Vorgängerknoten. Wir ergänzen dazu die Listenknoten um ein weiteres `Node`-Attribute `prev`.
Hier sehen Sie die beiden Listentypen skizziert:





Problem: Einfügen am Ende der Liste Das Einfügen am Ende der Liste erfordert zuerst das vollständige Durchlaufen der Liste. Dies ist bei Listen mit vielen Elementen nicht effizient.

- Lösung 1: Verwaltete Referenz auf das letzte Element der Liste (siehe Übung)
- Lösung 2: Füge ein Hilfselement ("Dummy") bei der doppeltverketteten Liste ein
  - Dessen `next`-Referenz zeigt auf den ersten Knoten der Liste.
  - Sein `prev`-Referenz zeigt auf das letzte Element der Liste.

  Diese Lösung vermeidet Spezialfälle bei der Implementierung der Methoden (z.B. beim Einfügen am Anfang der Liste).

2.3 Array-Listen

Statt mit Verkettungen zu arbeiten, kann man Listen auch mit Hilfe von Arrays implementieren. Die Elemente werden dabei intern in einem Array gespeichert. Dazu wird zunächst ein Array mit einer bestimmten Anfangsgröße initialisiert. Das Array wird dann sukkzessive mit neuen Elementen gefüllt. Das Attribut `size` verwaltet dabei, wie viele Elemente bereits hinzugefügt wurden. Wenn das Array zu klein für neue Elemente ist, wird ein größeres erstellt und die alten Elemente werden in das neue Array kopiert.

```
public class IntArrayList {
    private int[] elementData;
    private int size;

    public IntArrayList(int initialArrayLength) {
      elementData = new int[initialArrayLength];
      size = 0;
    }

    // Element am Ende der Liste einfuegen
    public void add(int element) {
        ensureCapacity(size + 1);
        elementData[size] = element;
        size++;
    }

    // stellt sicher, dass Array genug Platz hat
    private void ensureCapacity(int minCapacity) {
        if (elementData.length < minCapacity) {
            // Groesse verdoppeln, mindestens auf minCapacity
            int newSize = Math.max(minCapacity,
                                   2 * elementData.length);
            elementData = Arrays.copyOf(elementData, newSize);
        }
    }
```

Die Operationen müssen sicher stellen, dass nur der Teil des Arrays benutzt wird, der gültig ist. Um beispielsweise ein Element an einer bestimmten Indexposition zu erhalten, muss zunächst abgeprüft werden, ob dieser Index ein gültiges Listenelement enthält. Andernfalls wird ein Fehler ausgelöst (Details zur Fehlerauslösung im Kapitel zu Exceptions):

```
// liefert das Element an der gegebenen Position
public int get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return elementData[index];
}
```

Beim Suchen von Elementen wird nur über den gültigen Indexbereich iteriert:

```
// gibt die erste Position eines Elements in der Liste
//  oder -1, wenn das Element nicht in der Liste ist
public int indexOf(int element) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == element) {
            return i;
        }
    }
    return -1;
}
```

Zum Entfernen eines Elements aus der Array-Liste, wird zunächst das erste Vorkommen dieses Elements gesucht. Alles folgenden Elemente werden dann um eins nach links verschoben:

```
// entfernt das 1. Vorkommen eines Elements aus der Liste
public void remove(int element) {
  int indexOf = indexOf(element);
  if (indexOf < 0) {
      // Element nicht vorhanden
      return;
  }
  // alle Elemente nach dem ersten Vorkommen
  // nach links verschieben:
  for (int i = indexOf; i < size - 1; i++) {
      elementData[i] = elementData[i+1];
  }
  size--;
}
```

Das Löschen von Einträgen ist im Vergleich zu verketteten Listen aufwendig, insbesondere das Löschen am Anfang der Liste.

3 Iteratoren

Iteratoren erlauben es, schrittweise über sogenannte Datenstrukturen für Datenansammlungen (engl. *collections*) wie Listen zu laufen, so dass alle Elemente der Reihe nach besucht werden. Im Zusammenhang mit Kapselung sind sie unverzichtbar.
Ein Iterator stellt dabei zwei Methoden zur Verfügung:

- `hasNext()` prüft, ob es weitere Einträge gibt.
- `next()` liefert den nächsten Eintrag in der Datenansammlung.

Hier das Code-Gerüst für einen Iterator über eine Liste von `int`s:

```
public class IntListIterator {
    //prueft, ob es noch weitere Eintraege gibt
    public boolean hasNext(){ ... }

    //liefert den naechsten Eintrag
    public int next() {...}
}
```

Die Implementierung des Iterators hängt essentiell von der Implementierung der entsprechenden Datenansammlung ab. Im folgenden zeigen wir als Beispiel die Implementierung eines Iterators für einfachverkettete Listen.

```
  private Node current;
  public IntListIterator(Node e) {
    this.current = e;
  }
  //prueft, ob es noch weitere Eintraege gibt
  public boolean hasNext(){
    return current != null;
  }
  //liefert den naechsten Eintrag
  public int next() {
    int res = current.getValue();
    current = current.getNext();
    return res;
  }
}
```

Wir reichern nun die Klasse `IntList` mit Iteratoren an:

```
public class IntList {
  // Erweiterung um Iteratoren
  private Node first;
  ...
  public IntListIterator iterator() {
     return new IntListIterator(first);
  }
}
```

Jeder Aufruf von `iterator()` liefert einen neuen Iterator, mit dem über die Liste iteriert werden kann.

```
  public static void main(String[] args) {

    IntList l = new IntList();
    l.add(2);
    l.add(-7);
    l.add(34);

    IntListIterator iter = l.iterator();
    while(iter.hasNext()) {
      StdOut.println(iter.next());
    }
  }
}
```

Der Iterator muss Zugriff auf die interne Repräsentation der Datenstruktur haben, über die er iteriert. Der `IntListIterator` erhält daher die Referenz auf das erste Listenelement, `first`.
Wir zeigen als weiteres Beispiel eine Implementierung eines Iterators über eine Array-Liste. Dier erhält eine Referenz auf das interne Array der Array-Liste sowie die aktuelle Anzahl der Elemente. Das Attribut `position` gibt den Index des nächsten Elements an.

```
public class IntArrayListIterator {
    private int[] elementData;
    private int size;
    private int position;

    IntArrayListIterator(int[] elementData, int size) {
        this.elementData = elementData;
        this.size = size;
        this.position = 0;
    }

    public boolean hasNext() {
        return position < size;
    }

    public int next() {
        int elem = elementData[position];
        position++;
        return elem;
    }

}
```

Beim Erstellen des Iterators (in der Klasse `IntArrayList`) wird das interne Array an den Iterator übergeben:

```
    // liefert einen Iterator fuer die Liste
    public IntArrayListIterator iterator() {
        return new IntArrayListIterator(elementData, size);
    }
```

Frage 3: Was passiert, wenn der Array-Liste Elemente hinzugefügt bzw. entfernt werden, während der Iterator noch über die Liste iteriert?

Bemerkung: Eine alternative Möglichkeit, die Kapselung zu gewährleisten, sind innere Klassen, die wir im weiteren Verlauf der Vorlesung noch behandeln werden.