TU Berlin
Software and Business Engineering
Prof. Dr. Ingo Weber
Dr. Tobias Heindel
Richard Hobeck

WiSe 2021/22
Aufgabe 5
Block 2
22. November 2021

Aufgabenblatt 5 *Programmieren I*

Hinweise

- Die Abgabe dieser Übungsaufgaben muss bis spätestens Sonntag, den 28. November 2021 um 23:59 Uhr im ISIS-Kurs erfolgt sein. Es gelten die Ihnen bekannten Übungsbedingungen.
- Lösungen zu diesen Aufgaben sind als gezippter Projektordner abzugeben. Eine Anleitung zum Zippen von Projekten finden Sie auf der Seite des ISIS-Kurses. *Bitte benutzen Sie einen Dateinamen der Form* `VornameNachname.zip`.
- Bitte beachten Sie, dass Abgaben im Rahmen der Übungsleistung für die Zulassung zur Klausur relevant sind. Durch Plagiieren verwirken Sie sich die Möglichkeit zur Zulassung zur Klausur in diesem Semester.

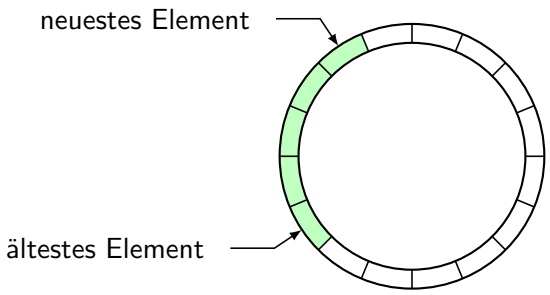

Abbildung 1: Pufferring

Aufgabe 5.1 *Abstrakte Klassen und ihre Instanzen: Pufferring* (2 Punkte)

Die Idee eines Pufferrings ist in Abbildung 1 gezeigt. Es können neue Elemente hinzugefügt werden, und zwar wird jedes neue Element in die nächste freie Zelle im Ring geschrieben (im Uhrzeigersinn); die einzige Ausnahme ist, wenn keine Zelle mehr frei ist. Wenn der Puffer nicht leer ist, gibt es also das zuletzt eingefügte Element und das älteste Element. Das Verhalten des Pufferrings ist in der abstrakten Klasse `AbstractBufferRing` implementiert (siehe Abbildung 2). Lediglich die konkrete Repräsentation der Zellen des Puffers ist nicht festgelegt.
Implementieren Sie eine Klasse `ObjectBufferRing`, die die Klasse `AbstractBufferRing` erweitert. Die Klasse `ObjectBufferRing` soll die `main`-Methode implementieren, in der Sie eine Instanz von `ObjectBufferRing` erzeugen, drei Objekte einfügen, und drei Objekte entfernen.

```
abstract public class AbstractBufferRing {

    private int frei;  // the "last" *free* cell (typically just before the "oldest" element)
    private int neueste; // most recently *used* cell (unless empty)
    private int groesse; // size of the ring
    private boolean full; // true if, and only if, all cells are used

    /**
     * initialize the common variables
     * according to the implementation on
     * the level of the abstract class
     * @param groesse the size of the buffer >= 1
     */
    protected AbstractBufferRing(int groesse){
        this.groesse = groesse;
        frei = 0;
        neueste = 0;
        full = false;
    }

    /**
     * check for emptiness
     * @return the truth value of "this buffer is empty"
     */
    public boolean isEmpty(){
        return frei == neueste && !full;
    }

    /**
     * check for fullness
     */
    public boolean isFull(){ return full; }


    /**
     * helper method for freeing the oldest cell
     */
    private void freeOldest(){
        frei++;
        frei = frei % groesse;
        full = false;
    }

    /**
     * helper method for advancing the index of the newest element
     */
    private void updateNeueste(){
        neueste++;
        neueste = neueste % groesse;
        if (neueste==frei) full=true;
    }

    /**
     * write an object at the cell of the index
     * @param o the object
     * @param index the index
     */
    abstract protected void writeToIndex(Object o, int index);

    /**
     * read an object at the cell of the index (if any)
     * @param index the index
     */
    abstract protected Object readFromIndex(int index);

    /**
     * put a new object into the buffer (if capacity admits)
     * @param o the new object to be put
     */
    public void put(Object o){
        if (!full){
            updateNeueste();
            writeToIndex(o, neueste);
        } else {
            System.out.println("Leider kein Platz mehr!");
        }
    }

    /**
     * remove the "oldest" object from the buffer
     * @return the "oldest" object (if any) or null (if none)
     */
    public Object remove(){
        if (isEmpty()) {
            System.out.println("Leider keine Elemente vorhanden!");
            return null;
        } else {
            // the following two lines should be atomic in a concurrent setting
            freeOldest();
            return readFromIndex(frei);
        }
    }

    public int getSize(){
        return this.groesse;
    }
}
```

Abbildung 2: Abstrakter Pufferring

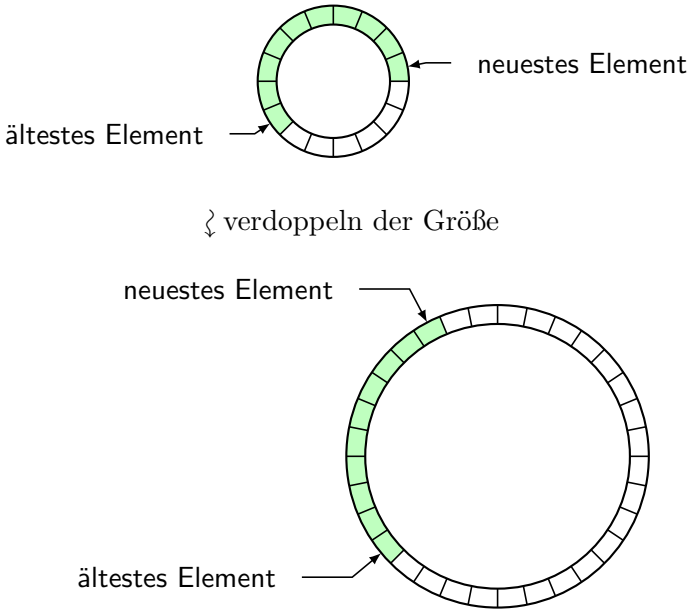

Abbildung 3: Erweiterbarer Pufferring

Aufgabe 5.2 *Implementieren von Interfaces: Erweiterbarer Pufferring* **(2 Punkte)**

Implementieren Sie eine Klasse `ExtendableBufferRing`, die

- einen Konstruktor `ExtendableBufferRing` mit einem `int`-Wert für die (initiale) Größe hat und
- das Interface `VariableBuffer` implementiert (siehe unten).

```
public interface VariableBuffer {
    void put(Object o);
    Object remove();
    boolean isEmpty();
    boolean isFull();
    void changeSizeBy(int size);
}
```

Die Methode `changeSizeBy` hat einen `int`-Wert als Formalparameter, der die Änderung der Größe des Puffers annimmt; das heißt, dass auch (sinnvolle) negative Werte berücksichtigt werden können.

Tipp Sie werden wahrscheinlich nicht direkt die abstrakte Klasse `AbstractBufferRing` erweitern wollen. Allerdings können Sie diese Klasse (bzw. die Lösung aus der ersten Teilaufgabe) dennoch verwenden.

Aufgabe 5.3 *Coding to an interface: automatisch wachsender Pufferring* **(1 Punkt)**

Als nächster logische Schritt folgt der Puffer, der „fast nie“ voll ist; die einzige Ausnahme sei, wenn es nicht genug Speicher gibt. Das heißt, der Puffer soll automatisch wachsen, wenn der Puffer voll ist, und zwar bis kein Speicher mehr verfügbar ist.
Implementieren Sie eine Klasse `UnlimitedBuffer`

- mit einem Konstruktor der als Formalparameter ein Objekt bekommt, welches das Interface `VariableBuffer` implementiert und
- einer `append`-Methode, mit einen `AbstractBufferRing`-Objekt als Formalparameter, dessen Inhalt zum UnlimitedBuffer-Objekt hinzugefügt wird.

Tipps

- Sie können das VariableBuffer-Objekt benutzen, um einen automatisch wachsenden Puffer zu implementieren.
- Eine gute Richtlinie ist die Größe des Puffers immer zu verdoppeln, wenn der Platz ausgeht.

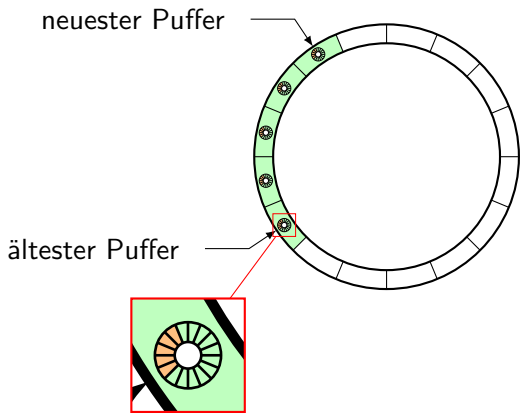

Abbildung 4: Pufferring von Pufferringn

Aufgabe 5.4 *Von Puffern von Puffern zu Puffern* **(1 Punkt)**

Schreiben Sie eine Klasse `FlattenBuffer` mit einer Klassenmethode `flatten`, die ein `AbstractBufferRing`-Objekt als Formalparameter hat, dessen Elemente wiederum alle `AbstractBufferRing`-Objekte sind. Ein solches Objekt ist in Abbildung 4 illustriert. Das heißt, Sie können annehmen, dass die `Object`-Objekte die von der `remove`-Methode zurückgegeben werden per `(AbstractBufferRing)` in den entsprechenden Typ umgewandelt werden können—Stichwort *type-casting*; die Ausgabe der Methode `flatten` soll ein neues `AbstractBufferRing`-Objekt sein, welches die Konkatenation aller Pufferring ist. Das Ergebnis für den Puffer von Puffern in Abbildung 4 wäre also ein Pufferring mit $5 \times 5 = 25$ Elementen wie in Abbildung 5

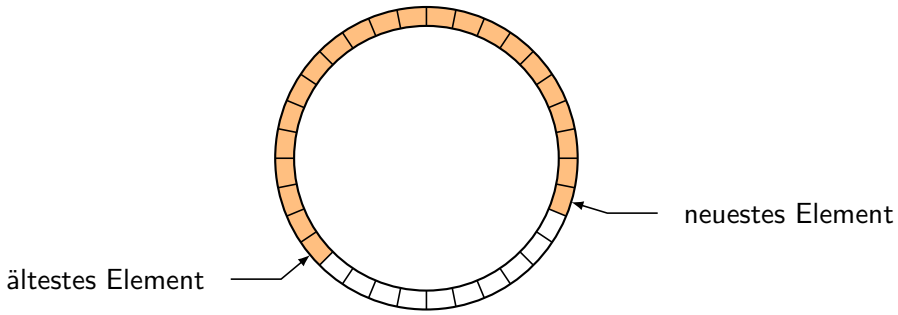

Abbildung 5: Die Konkatenation aller Puffer des Puffers von Puffern