Name: _____

Abiturprüfung 2016

Informatik, Leistungskurs

Aufgabenstellung:

Beim Transport von Waren werden oft Frachtcontainer in einer einheitlichen Größe eingesetzt. In einem Containerterminal werden solche Container nach ihrem Eintreffen für den weiteren Abtransport zwischengelagert. Um das Lagern in Containerterminals zu optimieren, soll eine Simulation des Lagerverfahrens modelliert und realisiert werden. Dabei wird das folgende Verfahren zugrunde gelegt:

Jedem Container ist eine Priorität zugewiesen. Die Priorität gibt an, ob ein Container voraussichtlich früher oder später abtransportiert wird. Je höher dieser Wert ist, desto früher wird er voraussichtlich abtransportiert. Es kann Container gleicher Priorität geben. Die Priorität ist eine positive ganze Zahl. Die Container werden auf einem geeigneten Platz übereinander gestapelt, wobei darauf geachtet wird, dass die Container eines Stapels gemäß ihrer Priorität sortiert sind. Container mit der höchsten Priorität in einem Stapel befinden sich immer oben.

In der Simulation soll es – wie in der Praxis – vorkommen können, dass verschiedene Plätze im Containerterminal zwischenzeitlich auch leer stehen. Eine Beschränkung der Höhe der einzelnen Containerstapel soll es hier nicht geben. Die Container werden ohne ein bestimmtes System auf die verschiedenen Plätze verteilt. Zur Vereinfachung soll der Containerterminal die feste Größe von fünf mal fünf Stellplätzen haben.

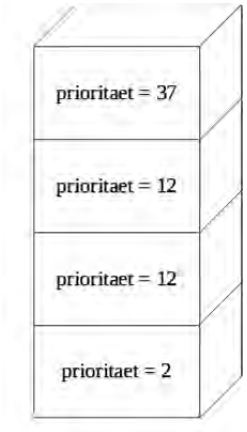

Abbildung 1:
Beispiel eines Containerstapels
(Ansicht von der Seite)

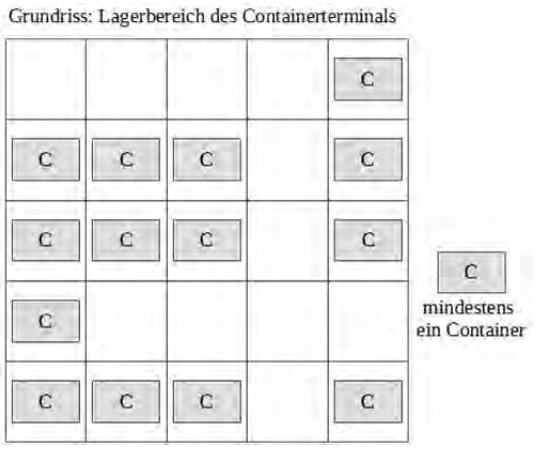

Abbildung 2:
Beispielbelegung eines Containerterminals
(Ansicht von oben)

Name: _____

Für eine objektorientierte Modellierung wird vorgeschlagen, dass jeder Container durch ein entsprechendes Objekt repräsentiert wird und diese Objekte in dynamischen Datenstrukturen organisiert werden. Die einzelnen Plätze im Containerterminal werden durch eine Felddatenstruktur (zweidimensionales Feld) verwaltet. Gesteuert wird die Simulation der Logistik über ein Objekt der Klasse `Containerterminal` (siehe Dokumentation im Anhang).

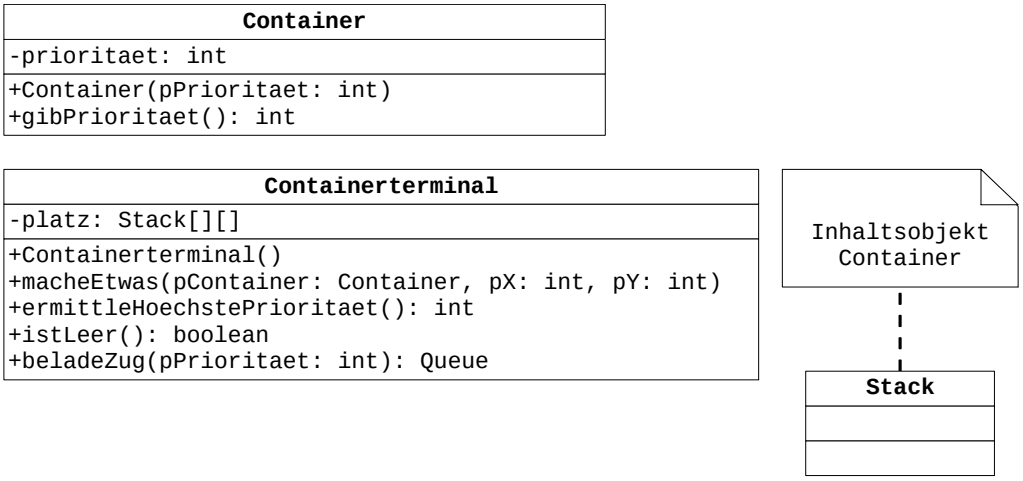

Abbildung 3: Implementationsdiagramm

a) *Beschreiben Sie die in Abbildung 3 dargestellten Klassen* `Container` *und* `Containerterminal` *und deren Beziehung zueinander.* (7 Punkte)

b) Die Methode `macheEtwas` der Klasse `Containerterminal` ist wie folgt implementiert:

```
 1 public void macheEtwas(Container pContainer, int pX, int pY) {
 2   Stack temp = new Stack();
 3   while (!platz[pX][pY].isEmpty()
          && ((Container) platz[pX][pY].top()).gibPrioritaet()
              > pContainer.gibPrioritaet()) {
 4     temp.push(platz[pX][pY].top());
 5     platz[pX][pY].pop();
 6   }
 7   platz[pX][pY].push(pContainer);
 8   while (!temp.isEmpty()) {
 9     platz[pX][pY].push(temp.top());
10     temp.pop();
11   }
12 }
```

Name: _____

Analysieren Sie die Methode `macheEtwas` *und erläutern Sie die Funktion der beiden Schleifen und der Variablen* `temp`.

Beschreiben Sie für jeden Parameter einen möglichen Fehler, der durch fehlerhafte Wertübergabe beim Aufruf zum Abbruch der Methode zur Laufzeit führt.

Erläutern Sie im Sachzusammenhang, was die Methode leistet.

(13 Punkte)

c) Das Beladen eines Zuges mit Containern soll simuliert werden. Dabei sollen alle Container verladen werden, deren Priorität größer oder gleich einem festgelegten Wert ist. Hierzu soll der Zug der Einfachheit halber durch ein Objekt der Klasse `Queue` simuliert werden. Die Container in der Schlangendatenstruktur sollen absteigend nach Priorität sortiert sein. Die Container mit der höchsten Priorität sollen zu Beginn eingeordnet sein.

Zur Simulation soll eine Methode `beladeZug` der Klasse `Containerterminal` entwickelt werden. Sie liefert als Ergebnis eine solche sortierte Schlangendatenstruktur. Die Methode hat den folgenden Methodenkopf:

```
public Queue beladeZug(int pPrioritaet)
```

Entwickeln Sie eine Strategie, mit der das Erstellen einer solchen Schlange gelöst werden kann.

Implementieren Sie die Methode `beladeZug`. *Die im Anhang dokumentierten Methoden können dabei verwendet werden.*

(12 Punkte)

Name: _____

In der Praxis schleichen sich im Containerterminal durch unsachgemäßes Stapeln der Container bisweilen Fehler ein: Die Container werden nicht entsprechend ihrer Priorität gestapelt. Die Auswirkungen von Fehlern dieser Art sollen in der Simulation untersucht werden.

In diesem Kontext wird eine alternative Modellierung vorgeschlagen: Für die Verwaltung der einzelnen Container auf einem Platz wird statt einer Stapeldatenstruktur eine Listendatenstruktur verwendet: Der Anfang der Liste soll dabei das obere Ende des Stapels repräsentieren.

d) Die Methode `ueberpruefePrioritaeten` der Klasse `Containerterminal` implementiert eine solche Überprüfung: Sie untersucht, ob die in einer als Parameter übergebenen Liste (Datentyp `List`) gespeicherten Container nach Priorität absteigend sortiert sind. Nur in diesem Fall wird der Wert `true` zurückgegeben. Eine leere Liste gilt als sortiert.

Vergleichen Sie, wie diese Art von Fehlern einerseits bei der Verwendung einer Listendatenstruktur und andererseits bei der Verwendung einer Stapeldatenstruktur erkannt werden kann.

Implementieren Sie die Methode `ueberpruefePrioritaeten`.

(10 Punkte)

e) Die Modellierung mit einer Liste repräsentiert die Struktur und das Verhalten des Containerstapels nur unzureichend. Es wäre wünschenswert, eine geeignetere Datenstruktur zu entwerfen. Bezüglich des Hinzufügens und Löschens der verwalteten Objekte sollte sie das Verhalten einer Stapeldatenstruktur aufweisen. Zusätzlich soll es aber möglich sein, die Datenstruktur zu durchlaufen und die Objekte einzusehen, ohne deren Reihenfolge in der Datenstruktur zu verändern.

Modellieren Sie eine solche Datenstruktur `InspectableStack`*, indem Sie die interne Verwaltung der Objekte erläutern und deren Methoden dokumentieren. Dokumentieren Sie nur die Abweichungen im Vergleich zur Klasse* `Stack`.

(8 Punkte)

Zugelassene Hilfsmittel:

- Wörterbuch zur deutschen Rechtschreibung
- Taschenrechner (wissenschaftlicher Taschenrechner ohne oder mit Grafikfähigkeit/ CAS-Taschenrechner)

Name: _____

Anhang

Die Klasse `Container`

Ein Objekt dieser Klasse verwaltet einen einzelnen Container mit seiner Priorität.

Ausschnitt aus der Dokumentation der Klasse `Container`

Konstruktor **`Container(int pPrioritaet)`**
Ein neues Objekt vom Typ `Container` wird erzeugt. Der Wert des Parameters `pPrioritaet` wird gespeichert.

Anfrage **`int gibPrioritaet()`**
Die Methode gibt die Priorität des Containers zurück.

Die Klasse Containerterminal

Ein Objekt dieser Klasse verwaltet alle 25 Plätze im Containerterminal, auf denen sich Container befinden können.

Ausschnitt aus der Dokumentation der Klasse `Containerterminal`

Konstruktor **`Containerterminal()`**
Ein neues Objekt vom Typ `Containerterminal` wird erzeugt. Es werden ein `5x5` Plätze großes Feld und die zugehörigen Stapelobjekte initialisiert.

Auftrag **`void macheEtwas(Container pContainer, int pX, int pY)`**
Die Methode wird in Aufgabenteil b) thematisiert.

Anfrage **`int ermittleHoechstePrioritaet()`**
Die Methode ermittelt die höchste Priorität unter allen Containern, die sich aktuell im Containerterminal befinden, und gibt diesen Wert zurück. Ist der Containerterminal leer, so wird `0` zurückgegeben.

Anfrage **`boolean istLeer()`**
Die Methode überprüft, ob sich noch mindestens ein Container im Containerterminal befindet, und gibt dann `false` zurück, sonst `true`.

Anfrage **`Queue beladeZug(int pPrioritaet)`**
Die Methode wird in Aufgabenteil c) thematisiert.

Name: _____

Die Klasse Stack

Objekte der Klasse **`Stack`** (Keller, Stapel) verwalten beliebige Objekte nach dem Last-In-First-Out-Prinzip, d. h., das zuletzt abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Stack

Konstruktor **`Stack()`**

Ein leerer Stapel wird erzeugt.

Anfrage **`boolean isEmpty()`**

Die Anfrage liefert den Wert `true`, wenn der Stapel keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag **`void push(Object pObject)`**

Das Objekt `pObject` wird oben auf den Stapel gelegt. Falls `pObject` gleich `null` ist, bleibt der Stapel unverändert.

Auftrag **`void pop()`**

Das zuletzt eingefügte Objekt wird von dem Stapel entfernt. Falls der Stapel leer ist, bleibt er unverändert.

Anfrage **`Object top()`**

Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird `null` zurückgegeben.

Name: _____

Dokumentation der Klasse Queue

Konstruktor `Queue()`

Eine leere Schlange wird erzeugt.

Anfrage `boolean isEmpty()`

Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag `void enqueue(Object pObject)`

Das Objekt `pObject` wird an die Schlange angehängt. Falls `pObject` gleich `null` ist, bleibt die Schlange unverändert.

Auftrag `void dequeue()`

Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.

Anfrage `Object front()`

Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird `null` zurückgegeben.

Name: _____

Die Klasse `List`

Objekte der Klasse **`List`** verwalten beliebig viele, linear angeordnete Objekte. Auf höchstens ein Listenobjekt, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.

Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt oder ein Listenobjekt an das Ende der Liste angefügt werden.

Dokumentation der Klasse `List`

Konstruktor **`List()`**
Eine leere Liste wird erzeugt.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert den Wert `true`, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert `false`.

Anfrage **`boolean hasAccess()`**
Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.

Auftrag **`void next()`**
Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d. h., `hasAccess()` liefert den Wert `false`.

Auftrag **`void toFirst()`**
Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Auftrag **`void toLast()`**
Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Name: _____

Anfrage **`Object getObject()`**

Falls es ein aktuelles Objekt gibt (`hasAccess()` == `true`), wird das aktuelle Objekt zurückgegeben, andernfalls (`hasAccess()` == `false`) gibt die Anfrage den Wert `null` zurück.

Auftrag **`void setObject(Object pObject)`**

Falls es ein aktuelles Objekt gibt (`hasAccess()` == `true`) und `pObject` ungleich `null` ist, wird das aktuelle Objekt durch `pObject` ersetzt. Sonst bleibt die Liste unverändert.

Auftrag **`void append(Object pObject)`**

Ein neues Objekt `pObject` wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt `pObject` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (`hasAccess()` == `false`). Falls `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag **`void insert(Object pObject)`**

Falls es ein aktuelles Objekt gibt (`hasAccess()` == `true`), wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (`hasAccess()` == `false`), wird `pObject` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt (`hasAccess()` == `false`) und die Liste nicht leer ist oder `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag **`void concat(List pList)`**

Die Liste `pList` wird an die Liste angehängt. Anschließend wird `pList` eine leere Liste. Das aktuelle Objekt bleibt unverändert. Falls `pList` `null` oder eine leere Liste ist, bleibt die Liste unverändert.

Auftrag **`void remove()`**

Falls es ein aktuelles Objekt gibt (`hasAccess()` == `true`), wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (`hasAccess()` == `false`). Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (`hasAccess()` == `false`), bleibt die Liste unverändert.

Unterlagen für die Lehrkraft

Abiturprüfung 2016

Informatik, Leistungskurs

1. Aufgabenart

Aufgabenart	Modellierung einer Problemstellung, Entwurf und Implementation von Algorithmen
Syntaxvariante	Java

2. Aufgabenstellung[1]

siehe Prüfungsaufgabe

3. Materialgrundlage

- entfällt

4. Bezüge zu den Vorgaben 2016

1. *Inhaltliche Schwerpunkte*
   Objektorientiertes Modellieren und Implementieren von kontextbezogenen Anwendungen
   - Konzepte des objektorientierten Modellierens
   - Algorithmen und Datenstrukturen
     - Lineare Strukturen mit den Akzenten
       Lineare Liste
       Schlange und Stapel

2. *Medien/Materialien*
   - entfällt

5. Zugelassene Hilfsmittel

- Wörterbuch zur deutschen Rechtschreibung
- Taschenrechner (wissenschaftlicher Taschenrechner ohne oder mit Grafikfähigkeit/ CAS-Taschenrechner)

[1] Die Aufgabenstellung deckt inhaltlich alle drei Anforderungsbereiche ab.

6. Modelllösungen

Die jeweilige Modelllösung stellt eine mögliche Lösung bzw. Lösungsskizze dar. Der gewählte Lösungsansatz und -weg der Schülerinnen und Schüler muss nicht identisch mit dem der Modelllösung sein. Sachlich richtige Alternativen werden mit entsprechender Punktzahl bewertet (Bewertungsbogen: Zeile „Sachlich richtige Lösungsalternative zur Modelllösung“).

Teilaufgabe a)

Ein Objekt der Klasse `Container` verfügt über eine Priorität, abgebildet durch das Attribut `prioritaet` vom Typ `int`. Dieses Attribut kann augenscheinlich auch über die Methode `gibPrioritaet` angefragt werden.

Ein Objekt der Klasse `Containerterminal` verfügt über die öffentliche Methode `wasMacheIch`, die drei Parameter hat und keine Rückgabe liefert. Außerdem bietet ein Objekt dieser Klasse die parameterlosen Anfragen `ermittleHoechstePrioritaet`, die einen Integerwert zurückgibt, und `istLeer`, die einen booleschen Wert zurückgibt, an. Die Methode `beladeZug` mit dem Parameter `pPrioritaet` liefert als Rückgabe ein Objekt der Klasse `Queue`.

Ein Objekt der Klasse `Containerterminal` verwaltet ein zweidimensionales Feld vom Typ `Stack` mit dem Namen `platz`. In dem durch `platz` bezeichneten Feld werden Containerstapel gespeichert. Objekte der Klasse `Container` sind Inhaltsobjekte der Stapel.

Teilaufgabe b)

Die Variable `temp` referenziert ein Stapelobjekt, auf dem alle Containerobjekte zwischengespeichert werden, die eine höhere Priorität als das einzufügende Objekt haben. Diese Objekte müssen nach dem Einsortieren des neuen Objekts wieder auf den ursprünglichen Stapel gelegt werden.

Die erste Schleife dient dazu, den Stapel so weit abzubauen, bis kein Objekt mehr im Stapel referenziert ist, das eine höhere Priorität als das einzufügende besitzt. Diese Objekte werden im temporären Stapel in umgekehrter Reihenfolge zwischengespeichert.

Die zweite Schleife dient dazu, alle Containerobjekte aus dem temporären Stapel wieder auf den ursprünglichen zu legen. Hierzu wird der temporäre Stapel wieder abgebaut, bis alle Objekte zurückgeschoben wurden.

Ein möglicher Fehler für den Parameter `pContainer` wäre, dass das zugehörige Objekt nicht existiert und damit eine Null-Referenz übergeben wird. Dies würde zu einem Fehler in Zeile 3 führen.

Ein möglicher Fehler für die Parameter `pX` und `pY` wäre, dass negative Werte übergeben werden und damit die Indizes beim Zugriff auf das Feld in Zeile 3 ungültig sind.

Die Methode legt eine Referenz auf das als Parameter übergebene Objekt vom Typ `Container` gemäß seiner Priorität an der richtigen Stelle des Stapels ab. Objekte mit höherer Priorität befinden sich am Ende des Einfügeprozesses oberhalb, Objekte mit niedrigerer oder gleicher Priorität unterhalb des eingefügten Objektes.

Teilaufgabe c)

Es muss zunächst (z. B. mithilfe der Methode `ermittleHoechstePrioritaet`) die höchste Priorität aller noch im Containerterminal befindlichen Container bestimmt werden.

Da die Container in den einzelnen Stapeln nach der Priorität sortiert sind, müssen nun zunächst die jeweils obersten Elemente jedes Stapels überprüft werden.

Handelt es sich um einen Container mit der bereits ermittelten höchsten Priorität, wird dieser in die Schlangenstruktur eingeführt und aus dem jeweiligen Stapel entfernt.

Nach einem kompletten Durchlauf durch alle obersten Elemente der Stapelobjekte beginnt das Verfahren von vorne und läuft so lange, bis alle Containerobjekte entfernt wurden.

```
public Queue beladeZug(int pPrioritaet) {
  Queue zug = new Queue();

  /* Ermitteln der hoechsten vorhandenen Prioritaet. */
  int max = ermittleHoechstePrioritaet();
  while(!istLeer() && max >= pPrioritaet) {

    /* Laufe alle Plaetze durch. */
    for (int i = 0; i < platz.length; i++) {
      for (int j = 0; j < platz[i].length; j++) {

        /* Wenn der Platz nicht leer ist. */
        if (!platz[i][j].isEmpty()) {
          Container aktuell = (Container) platz[i][j].top();

          /* Wenn das Element das gesuchte ist,
           * so raeume es aus. */
          if (aktuell.gibPrioritaet() == max) {
            zug.enqueue(aktuell);
            platz[i][j].pop();
          }
        }
      }
    }
    max = ermittleHoechstePrioritaet();
  }
  return zug;
}
```

Teilaufgabe d)

Da ein Stapel nur Zugriff auf das oberste Objekt hat, müsste man, um eine solche Art von Fehlern zu erkennen, alle Stapel komplett ab- und wieder aufbauen, um währenddessen untersuchen zu können, ob die enthaltenen Objekte in der korrekten Reihenfolge sind.

Mithilfe der Listenstruktur kann man auf jedes Objekt der Liste zugreifen. Hier ist es also möglich, die Priorität für jedes Objekt im Containerstapel zu ermitteln, ohne die eigentliche Struktur zu verändern. Damit kann also überprüft werden, ob sich die Container mit absteigender Sortierung der Priorität in der Liste befinden. Hierzu muss jede Liste von vorne durchlaufen werden und die Priorität eines Containers mit der seines Vorgängers (oder Nachfolgers) verglichen werden. Sollte die Priorität eines Containers größer sein als die seines Vorgängers, ist eine fehlerhafte Stelle identifiziert.

```
public boolean ueberpruefePrioritaeten(List pListe) {
  int prioVorgaenger;
  Container folgender;
  pListe.toFirst();
  boolean ergebnis = true;

  /* Einen Vorgaenger des ersten Containers gibt es nicht,
   * daher eine Sonderbehandlung. */
  if (pListe.hasAccess()) {
    folgender = (Container) pListe.getObject();
    prioVorgaenger = folgender.gibPrioritaet();
    pListe.next();
    while (pListe.hasAccess()) {

      /* Merke dir den folgenden Container. */
      folgender = (Container) pListe.getObject();

      /* Wenn der folgende Container eine hoehere Prioriaet hat,
       * ist etwas falsch. */
      if (folgender.gibPrioritaet() > prioVorgaenger) {
        ergebnis = false;
      }

      /* Um alle zu testen wird der erste Container
       * zum zweiten (usw.). */
      prioVorgaenger = folgender.gibPrioritaet();
      pListe.next();
    }
  }
  return ergebnis;
}
```

Teilaufgabe e)

Die interne Verwaltung der Objekte kann ähnlich wie in einer Liste vorgenommen werden: Alle Inhaltsobjekte sind in einem Knotenobjekt gekapselt. Das Knotenobjekt verwaltet die Referenz auf das Inhaltsobjekt sowie eine Referenz auf das nächste Knotenobjekt. Liegt unter dem Knotenobjekt im Stapel kein Objekt, verweist diese Referenz auf `null`.

Ein Objekt der Klasse `InspectableStack` referenziert nun jeweils das oberste Knotenobjekt des Stapels und zusätzlich ein aktuelles Knotenobjekt, mit dem – ähnlich wie in einer Liste – der Stapel nach unten durchlaufen und auf die inneren Objekte zugegriffen werden kann.

Die Methoden `isEmpty`, `push` und `top` müssen nicht dokumentiert werden!

Konstruktor **`InspectableStack()`**

Ein leerer inspizierbarer Stapel wird erzeugt.

Auftrag **`void pop()`**

Das zuletzt eingefügte Objekt wird von dem Stapel entfernt. Falls der Stapel leer ist, bleibt er unverändert. Wird das aktuelle Element entfernt, gibt es kein aktuelles Objekt mehr.

Anfrage **`boolean hasAccess()`**

Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.

Auftrag **`void down()`**

Falls der inspizierbare Stapel nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das unterste Objekt des Stapels ist, wird das dem aktuellen Objekt im Stapel darunterliegende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d. h., `hasAccess()` liefert den Wert `false`.

Auftrag **`void toTop()`**

Falls der inspizierbare Stapel nicht leer ist, wird das erste Objekt des Stapels aktuelles Objekt. Ist der Stapel leer, geschieht nichts.

Anfrage **`Object getObject()`**

Sofern es ein aktuelles Objekt gibt (`hasAccess`), wird eine Referenz auf das Objekt zurückgegeben. Andernfalls wird `null` zurückgegeben.

7. Teilleistungen – Kriterien / Bewertungsbogen zur Prüfungsarbeit

Name des Prüflings:_____ Kursbezeichnung:_____

Schule: _____

Teilaufgabe a)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**[2]	**ZK**	**DK**
1	beschreibt die Klasse `Containerterminal`.	2			
2	beschreibt die Klasse `Container`.	2			
3	beschreibt die Beziehung der beiden Klassen zueinander.	3			
Sachlich richtige Lösungsalternative zur Modelllösung: (7) …… ……					
	Summe Teilaufgabe a)	**7**			

Teilaufgabe b)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	erläutert die Funktion der beiden Schleifen und der Variablen `temp`.	6			
2	beschreibt je einen möglichen Laufzeitfehler.	4			
3	erläutert im Sachzusammenhang, was die Methode leistet.	3			
Sachlich richtige Lösungsalternative zur Modelllösung: (13) …… ……					
	Summe Teilaufgabe b)	**13**			

[2] EK = Erstkorrektur; ZK = Zweitkorrektur; DK = Drittkorrektur

Teilaufgabe c)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	entwickelt die Strategie zum Suchen von Containern der aktuell höchsten Priorität in den obersten Elementen aller Stapel.	3			
2	entwickelt eine Strategie zum Einfügen in die Liste / Entfernen aus dem Stapel.	2			
3	implementiert das Erzeugen der Schlange und das Überführen der Objekte in die Schlange.	4			
4	implementiert den Abbau der Containerstapel.	3			
Sachlich richtige Lösungsalternative zur Modelllösung: (12) …… ……					
	Summe Teilaufgabe c)	**12**			

Teilaufgabe d)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	vergleicht, wie diese Art von Fehlern erkannt werden kann.	4			
2	implementiert die Methode.	6			
Sachlich richtige Lösungsalternative zur Modelllösung: (10) …… ……					
	Summe Teilaufgabe d)	**10**			

Teilaufgabe e)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	modelliert die Datenstruktur und dokumentiert die Methoden.	4			
2	erläutert die interne Verwaltung der Objekte.	4			
Sachlich richtige Lösungsalternative zur Modelllösung: (8) …… ……					
	Summe Teilaufgabe e)	**8**			

	Summe insgesamt	**50**			

Festlegung der Gesamtnote (Bitte nur bei der letzten bearbeiteten Aufgabe ausfüllen.)

	Lösungsqualität			
	maximal erreichbare Punktzahl	EK	ZK	DK
Übertrag der Punktsumme aus der ersten bearbeiteten Aufgabe	**50**			
Übertrag der Punktsumme aus der zweiten bearbeiteten Aufgabe	**50**			
Übertrag der Punktsumme aus der dritten bearbeiteten Aufgabe	**50**			
Punktzahl der gesamten Prüfungsleistung	**150**			
aus der Punktsumme resultierende Note gemäß nachfolgender Tabelle				
Note ggf. unter Absenkung um bis zu zwei Notenpunkte gemäß § 13 Abs. 2 APO-GOSt				
Paraphe				

Berechnung der Endnote nach Anlage 4 der Abiturverfügung auf der Grundlage von § 34 APO-GOSt

Die Klausur wird abschließend mit der Note _____ (____ Punkte) bewertet.

Unterschrift, Datum:

Grundsätze für die Bewertung (Notenfindung)

Für die Zuordnung der Notenstufen zu den Punktzahlen ist folgende Tabelle zu verwenden:

Note	**Punkte**	**Erreichte Punktzahl**
sehr gut plus	15	150 – 143
sehr gut	14	142 – 135
sehr gut minus	13	134 – 128
gut plus	12	127 – 120
gut	11	119 – 113
gut minus	10	112 – 105
befriedigend plus	9	104 – 98
befriedigend	8	97 – 90
befriedigend minus	7	89 – 83
ausreichend plus	6	82 – 75
ausreichend	5	74 – 68
ausreichend minus	4	67 – 60
mangelhaft plus	3	59 – 50
mangelhaft	2	49 – 40
mangelhaft minus	1	39 – 30
ungenügend	0	29 – 0