AD - Aufgabe 1 - Entwurf

Inhaltsverzeichnis

AD - Aufgabe 1 - Entwurf

Package-/Ordner-Hierarchie und Sichtbarkeiten

adt
----- interface
---------- AdtContainer <public>
---------- AdtList <public>
---------- AdtStack <public>
---------- AdtQueue <public>
---------- AdtArray <public>
----- implementations
---------- AdtContainerFactory <public>
---------- AdtListImpl <package private>
---------- AdtStackImpl <package private>
---------- AdtQueueImpl <package private>
---------- AdtArrayImpl <package private>
---------- tests
---------- AdtListTest <public>
---------- AdtStackTest <public>
---------- AdtQueueTest <public>
---------- AdtArrayTest <public>
---------- AdtAllTests <public>

Klassen/interface Hierarchien und Sichtbarkeiten

AdtContainer
----- AdtList
---------- AdtListImpl
----- AdtStack
---------- AdtStackImpl
----- AdtQueue
---------- AdtQueueImpl
----- AdtArray
---------- AdtArrayImpl

Alle anderen Klassen haben keine Elternklasse, bzw. erben nur von “object”.

AD - Aufgabe 1 - Entwurf

Interfaces/Spezifikationen/Verträge

AdtContainer

- dient lediglich als Marker interface für die Hierarchie und gibt somit keine Operationen vor

Folgende Operationen werden von den interfaces als “public” vorgegeben:

AdtList

- isEmpty (input: void | output: boolean isEmpty)
- length (input: void | output: int length) { entspricht der vorgegebenen Operation “laenge” }
- insert (input: int pos, int elem | output: void)
- delete (intput: int pos | output: void)
- find (input: int elem | output: int pos)
- retrieve (input: int pos | output: int elem)
- concat (input: AdtList list | output: void)

AdtStack

- push (input: int elem | output: void)
- pop (input: void | output: void)
- top (input: void | output: int elem)
- isEmpty (input: void | output: boolean isEmpty)

AdtQueue

- front (input: void | output: int elem)
- enQueue (input: Integer elem | output: void) { entspricht “enqueu” }
- deQueue (input: void | output: void)
- isEmpty (input: void | output: boolean isEmpty)

AdtArray

- set (input: int pos, int elem | output: void) { entspricht “setA” }
- get (input: int pos | output: int elem) { entspricht “getA” }
- length (input: void | output: int length) { entspricht “lengthA” }

AD - Aufgabe 1 - Entwurf

AdtContainerFactory

- enthält für jede Implementation eine statische Methode zum erstellen dieser, wobei als Rückgabetyp das interface gewählt wird.
(Methoden: adtList(), adtStack(), adtQueue(), adtArray() { entsprechen den create/initA methoden })

Die Liste ADT's sind allesamt mutable, angesichts dessen, dass immutable Container recht ineffizient sind, vor allem in einer Sprache, die nicht auf die funktionale Programmierung ausgelegt ist.

Abstraktionsschichten

1) Die Implementationen enthalten allesamt die statische Methode "valueOf" { entspricht der vorgegebenen Operation "create" }. Der Konstruktor ist von außen nicht zugänglich, wodurch man nicht an diesen gebunden ist und gegebenenfalls Techniken wie z.B. "Pooling" einführen kann.
2) Die Implementationen sind, wie aus der Package-Hierarchie zu entnehmen ist, nur innerhalb des Pakets sichtbar und von außen nur erstellbar durch die Klasse "AdtContainerFactory", die sich im selben Package befindet. Dadurch sind die Implementationen austauschbar, ohne das man von außen etwas davon mitbekommt.
3) Als Rückgabetyp wird immer das Interface gewählt, sodass man immer nur an den von diesem vorgegebenen Vertrag gebunden ist und niemals an eine konkrete Implementation.

Anmerkungen zu den Implementationen

Es muss daran gedacht werden "equals" in jeder Implementation zu überschreiben, damit nicht nach Referenz-, sondern nach Wertgleichheit geprüft wird.
Anderenfalls werden die Tests nicht funktionieren.

Beispiel:
```
AdtList adtList1 = AdtContainerFactory.adtList();
AdtList adtList2 = AdtContainerFactory.adtList();
```

AD - Aufgabe 1 - Entwurf

```
for (int i = 1; i < 5; i++) adtList1.insert(i, i);
for (int i = 1; i < 5; i++) adtList2.insert(i, i);

System.out.println(adtList1 == adtList2); // -> false
System.out.println(adtList1.equals(adtList2); // sollte true sein, basiert aber
                standartmäßig auf == und muss somit überschrieben werden
```

Bei AdtStack und bei AdtQueue muss zusätzlich darauf geachtet werden, dass die Iteration durch diese imperativ verläuft, “equals” aber keine imperative Operation darstellen darf.

AdtListImpl

- Intern wird für das Speichern der Liste ein Java Array verwendet (genauer: int[]), woraus sich auch der Typ des einzufügenden Elements ergibt
- Eine leere Liste hat die Länge 0
- Die erste Position an der man ein Element einfügen kann, ist 1
- Man kann immer zwischen Position 1 und Position length() + 1 ein Element einfügen (einschließlich)
- “find”, “retrieve” gibt 0 beim Fehlschlag zurück
- “insert”, “delete” verändern die Liste nicht, bei einem Fehlschlag
- “insert”, “delete”, “concat” haben keinen Rückgabewert und arbeiten imperativ, verändern also den Receiver
- Beim erfolgreichen Einfügen an einer Stelle, an der bereits ein Element steht, wird dieses, wie auch alle nachfolgenden Elemente, um 1 nach rechts verschoben

AdtStackImpl

- Intern wird die AdtList als Speicher verwendet, woraus sich der Typ für die Elemente ergibt
- “top” gibt 0 beim Fehlschlag zurück
- “push”, “pop” haben keinen Rückgabewert und arbeiten imperativ, verändern also den Receiver

AdtQueueImpl

- Basiert intern auf zwei AdtStack’s. Einen für den Input und einen für den Output, wobei an umstapeln zwischen den beiden Stack’s an passender Stelle gedacht werden muss.

- “front” gibt 0 beim Fehlschlag zurück
- “push”, “pop” haben keinen Rückgabewert und arbeiten imperativ, verändern also den Receiver

AdtArrayImpl

- Die erste Position des Array’s ist die 0 und zu diesem Zeitpunkt ist die Länge “-1”
- Das Einfügen eines Elements an Position “x” in das Array mit der Länge “y” setzt die Länge des Array’s auf “x”, sofern gilt: x > y
- Man kann ein Element an eine beliebige Stelle, solange diese > 0 ist, einfügen
- Das Einfügen eines Elements an eine Position, an der bereits ein Element steht, führt zur Überschreibung
- “get” gibt 0 beim Fehlschlag, oder/bzw. beim Zugriff auf eine nicht gesetzte Stelle, zurück
- “set” hat keinen Rückgabewert und arbeitet imperativ, verändert also den Receiver

Zusätzlich sei zu allen Implementationen angemerkt, dass diese eine theoretisch unendliche (praktisch durch den Computer-Speicher begrenzte) Anzahl an Elementen aufnehmen können. Da die AdtListImpl intern auf eine Java Array aufbaut, die eine fixe Größe hat, muss hier entsprechend darauf geachtet werden.

Tests

Es sind alle öffentlich, durch das Interface vorgegebenen Operationen zu testen und zusätzlich ist es wichtig, “equals” zu testen.

Zu testende Fälle sind:
- Leere Datenstrukturen
- Mittlere Datenstrukturen
- Große Datenstrukturen mit 1.000 und mehr Einträgen (vor allem da man letzendlich auf die Array von Java aufbaut und diese eine fixe Größe hat. Die Vergrößerung dieser muss getestet werden!)
- Datenstruktur mit sich wiederhohlenden Daten
- Operationen wie z.B. “enQueue” und “deQueue” stehen sich gegenüber und erzeugen praktisch entgegengesetzte Ergebnisse. Dies muss getestet werden
- Sofern entsprechende Operation enthalten ist, muss das Einfügen und löschen von Daten am Anfang, in der Mitte, am Ende und außerhalb (zu kleiner und/oder zu großer Index) getestet werden

AD - Aufgabe 1 - Entwurf

Bei den Test’s muss insbesondere darauf geachtet werden, dass es unter normalen Umständen keinesfalls zu Fehlermeldungen kommen wird, sondern dass stattdessen entweder keine Änderungen durchgeführt werden, oder man eine 0 als Rückgabe erhält. Vor allem bei der 0 muss unterschieden werden zwischen der 0 als Fehler und der gewünschten 0 als tatsächlich enthaltender Wert, sofern man diesen eingetragen hat. Die 0 ist für die Tests ohnehin interessant und darf somit nicht fehlen.

Export

An die zu exportierenden Dateien ist noch “ad_” vorweg zu hängen, zur besseren Unterscheidung mit adt’s anderer Veranstaltungen.

Alle ADT’s zusammen: ad_adt.jar
Alle Tests zusammen: ad_adtTests.jar
Einzeln jeweils den Klassennamen nach, aber beginnend mit Kleinbuchstaben: ad_adtListImpl.jar, ad_adtQueueImpl.jar ...
Bei den Tests ebenfalls: ad_adtListTest.jar, ad_adtQueueTest.jar ...

Die *.jar Dateien können dann als Bibliotheken von anderen eingebunden werden und solange man in den Dateien die gleiche Ordner Struktur hat (also so wie hier vereinbart), sind keine Änderungen durchzuführen. Die Test’s/Implementationen funktionieren nach dem Importieren ohne jegliche Änderungen.