


 **Artemis** 6.4.0

[Course Overview](#)



 ge64baw

[Courses](#) > [Praktikum: Grundlagen der Programmierung WS22/23](#) > [Exercises](#) > [W05H03 - A Messenger is Still on the List](#)

 **W05H03 - A Messenger is Still on the List**

Hausaufgabe

Hard

Submission due:


8 months ago



Points: 6 of 6

Assessment: automatic ?

Complaint due:

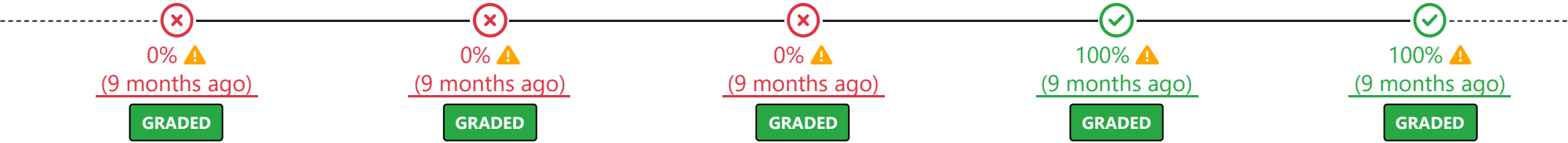
8 months ago

 Resume practice in exercise

 100%  (9 months ago)

GRADED

Recent results:



Show all results

Tasks:

W05H03 - Ein Messenger steht noch auf der Liste

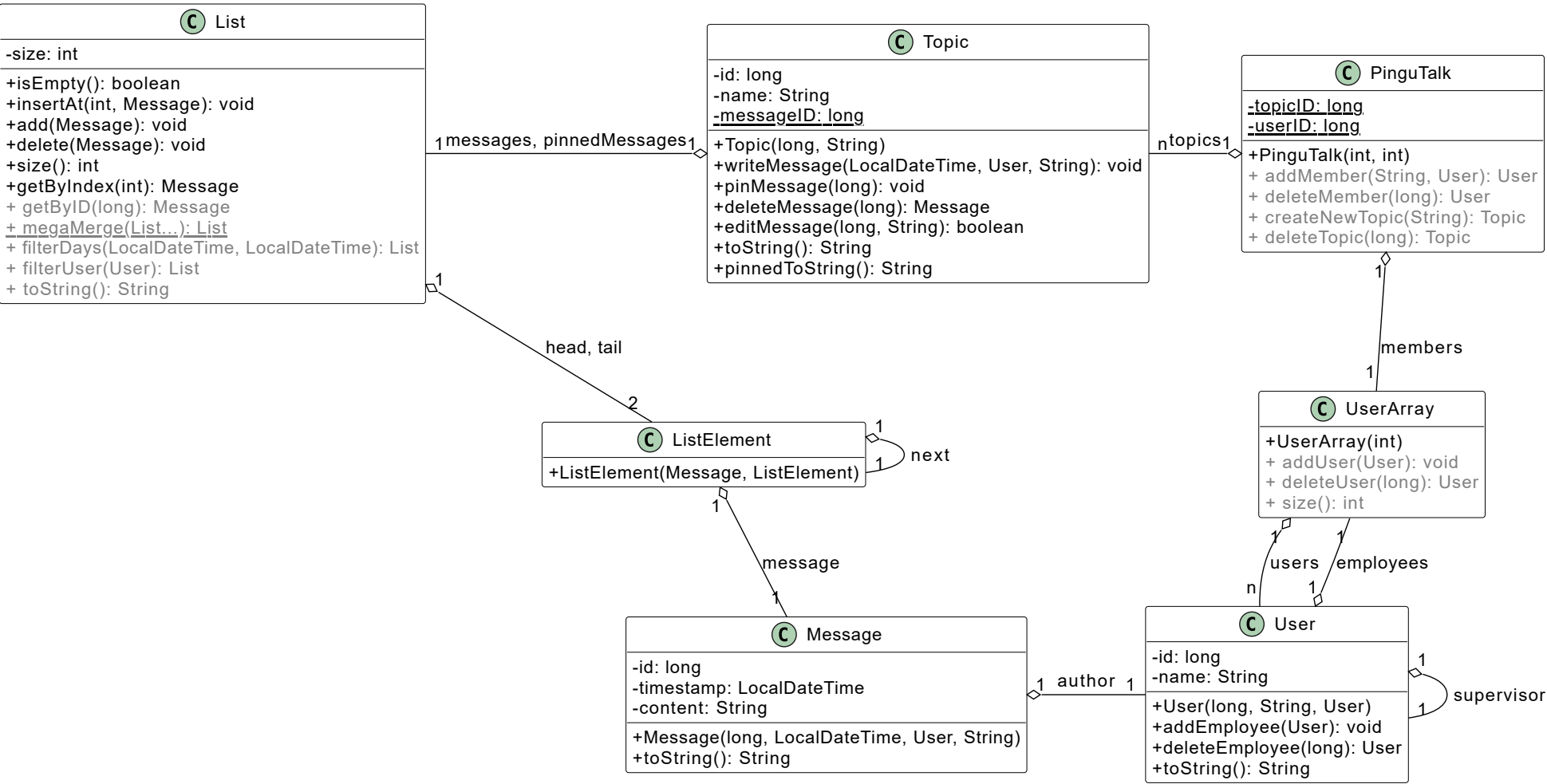
Der Saisonstart der Formel 1 rückt näher und unser Team Alfa Südpoleo ist schon Dank deiner Tests gut aufgestellt. Leider muss noch ein schwerwiegendes Problem gelöst werden: Teamchef Günther Steinbock hat immer noch keine gute Möglichkeit, seine Mitarbeiterinnen und Mitarbeiter zu kontaktieren. Sein Fahrer Valtteri Ottas ist im Urlaub in Spa und ebenfalls unerreichbar. Ohne seine rauen Anweisungen können die Raceuine nicht arbeiten. Da du die letzte Aufgabe so toll gelöst hast, wurdest du beauftragt, einen Text-Messenger zu programmieren, welcher auf allen Handys zwangsinstalliert werden soll. Einen Namen haben sich die Raceuine auch schon überlegt: Pingu Talk. Sie haben bereits ein UML-Diagramm entworfen, welches die Umsetzung des Messengers beschreibt, sowie eine Spezifikation der einzelnen Klassen. Und sie haben sogar schon mit dem Coden angefangen, hängen nun allerdings noch an einigen der schwierigeren Teile. Kannst du ihnen weiterhelfen?

Allgemeine Hinweise:

- Zu jeder Klasse werden die Behavioral Tests der Methoden (also die Tests, die das Verhalten der jeweiligen Methode überprüfen) erst ausgeführt, wenn der zugehörige Structural Test der Klasse (also der Test, welcher überprüft, ob bspw die Methoden vorhanden sind und die korrekte Signaturen tragen) durchläuft. Das schließt alle vorhandenen Attribute und Methoden ein, sowie funktionierende Getter und Setter und den Konstruktor.
- Die Klasse `LocalDateTime` entstammt dem Package `java.time`. Sie ist eine bereits vom JDK zur Verfügung gestellte Klasse und repräsentiert einen Zeitpunkt bestehend aus einem Datum und einer Uhrzeit. [Hier](#) findest du die Dokumentation. Methoden, die für dich nützlich sein könnten, sind `LocalDateTime.isBefore(LocalDateTime)`, `LocalDateTime.isEqual(LocalDateTime)` und `LocalDateTime.isAfter(LocalDateTime)`.
- Das komplette Package `java.util` ist für diese Aufgabe verboten. Wer es trotzdem verwenden will, macht die Raceuine traurig :(und die FIA kommt vorbei und wird nicht nur deine Factory schließen ...
- Achte auch in dieser Aufgabe wieder auf die Prinzipien der Datenkapselung, wie du sie in der Zentralübung kennengelernt hast.
- Die Public Tests testen die Struktur der Klassen (also Signaturen der Methoden, Attribute, etc) sowie jede Methode anhand eines einfachen Beispiels.

Diagramm:

Das Programm soll folgendermaßen aufgebaut sein. Beachte, dass Getter und Setter nicht explizit im Diagramm aufgeführt werden, aber dennoch grundsätzlich implementiert werden sollen (außer es steht weiter unten etwas anderes):



Kurzbeschreibung der einzelnen Klassen.

(Einige der hier aufgelisteten Klassen/Funktionalitäten sind bereits implementiert. Wir stellen dennoch eine vollständige Liste zur Verfügung, sodass du einen Überblick hast, was die einzelnen Komponenten des Programms tun.)

UserArray

Die Klasse `UserArray` ist ein Wrapper für das `User[]` (für diese Teilaufgabe gilt im Folgenden `User[] = Array`). Ein neues `UserArray` soll mit der übergebenen Startlänge initialisiert werden. Dabei soll das `Array` immer mindestens mit der Länge `1` initialisiert werden. Außerdem soll die Klasse Folgendes können:

- `addUser()` soll den übergebenen `User` an der ersten freien Stelle im `Array` speichern. Falls das `Array` voll ist, so soll vor dem Speichern die Größe des `Arrays` verdoppelt werden. Wenn der `User` `null` ist, soll nichts passieren.
- `deleteUser()` soll den zur übergebenen `id` passenden `User` finden, aus dem `Array` entfernen und zurückgeben. Die dabei entstandene Lücke darf bleiben. Falls kein `User` zu der `id` passt, soll `null` zurückgegeben werden.
- `size()` gibt die Anzahl der gespeicherten `User` zurück.

User

Diese Klasse ist bereits implementiert und verwaltet Informationen zu einem `User`. Ein neuer `User` wird mit einer übergebenen `id`, einem `name` und einem `supervisor` initialisiert, und initialisiert `employees` als `UserArray` der Größe `10`. Außerdem kann die Klasse Folgendes:

- `addEmployee()` bekommt einen `user` übergeben und fügt diesen in das `UserArray` ein.
- `deleteEmployee()` löscht den zur übergebenen `id` passenden `User` aus dem `UserArray`.
- `toString()` gibt eine `String`-Repräsentation des `Users` nach folgendem Schema zurück: "`<User-ID> ; <User-Name>`".

Message

Diese Klasse ist bereits implementiert und verwaltet Informationen zu einer `Message`. Eine neue `Message` wird mit einer übergebenen `id`, einem `timestamp`, einem `author` und einem `content` initialisiert. Außerdem kann die Klasse Folgendes:

- `toString()` gibt eine `String`-Repräsentation der `Message` nach folgendem Schema zurück: "`<Message-ID> ; <User-ID> ; <Timestamp-String> ; <Message-Content>`". Falls der `author` nicht existiert, wird die durch "`No_Author_Available`" ersetzt. Falls der `timestamp` nicht existiert, wird der durch "`No_Time_Available`" ersetzt. Die anderen Bestandteile des Strings bleiben ansonsten gleich.

List

Die `Liste` benutzen wir, um `Messages` eines Chats zu speichern. Die Implementierung sollte dir schon grob aus der P-Aufgabe bekannt vorkommen. Unsere `Liste` besteht aus einer Kette einzelner `ListElements`. Dabei wird für die `Liste` der Default-Konstruktor verwendet und Getter/Setter werden gar nicht benötigt. Deine Aufgabe ist es nun die Funktionalität um fünf kleine Methoden zu erweitern. Ändere die vorgegebenen Methoden daher **nicht**:

- `getByID()` gibt zur übergebenen `id` die zugehörige `Message` zurück. Falls die `Message` nicht existiert, soll `null` zurückgegeben werden.
- `megaMerge()` ist eine statische Methode und nimmt beliebig viele nach `timestamp` aufsteigend sortierte `List`en an (auch gar keine sind möglich), welche unterschiedlich lang sein können. Zurück gibt sie eine `List`e, in der alle `Messages` aller `List`en sortiert gemergt wurden. Was mit den Input-

`Listen` passiert bleibt dir überlassen. Falls keine `Liste` übergeben werden, so ist das Ergebnis eine leere `Liste`.

- `filterDays()` bekommt zwei `LocalDateTimes` (`start` und `end`) übergeben. Sie gibt eine neue `Liste` zurück, die die Nachrichten in selber Reihenfolge wie die ursprüngliche sortierte `Liste` enthält. Aber zusätzlich müssen diese `Messages` zwischen `start` (inklusive) und `end` (exklusiv) liegen. Falls einer der Parameter `null` ist oder `end` vor `start` liegt, soll eine leere `Liste` zurückgegeben werden. Die ursprüngliche `Liste` darf dabei nicht verändert werden.
- `filterUser()` funktioniert genauso wie `filterDays()` mit dem Unterschied, dass wir jetzt nach dem übergebenen `User` filtern. Achte auch hier darauf, die ursprüngliche `Liste` nicht zu verändern.
- `toString()` eine String-Repräsentation der `Liste` und gibt diese zurück, indem einfach die String-Repräsentationen der `Messages` der einzelnen `ListElements` getrennt durch ein Zeilenumbruch konkateniert werden (Zeilenumbruch auch am Ende). Falls die Liste leer ist, soll ein leerer `String` zurückgegeben werden.

Topic

Diese Klasse ist bereits implementiert und verwaltet einen Chat. Ein neues `Topic` wird mit einer übergebenen `id` und einem `name` initialisiert, und initialisiert die beiden `Listen` `messages` und `pinnedMessages`. Mit `messageID` soll nichts passieren.

- Das statische Attribut `messageID` sorgt dafür, dass jede `Message` auch über mehrere `Topics` hinweg eine eindeutige `id` bekommt. Einzig `messageID` bekommt weder Getter noch Setter.
- `writeMessage()` erstellt aus der aktuellen `messageID`, sowie den übergebenen Parametern `timestamp`, `author` und `content` ein neues `Message`-Objekt und fügt das in die `messages`-Liste ein. Nach jeder neuen `Message` wird die `messageID` inkrementiert.
- `pinMessage()` sucht in `messages` nach der zur `id` gehörenden `Message` und fügt diese in die `pinnedMessages`-Liste ein, ohne neue `Message`-Objekt dabei zu erzeugen. Falls die Message nicht existiert, wird nichts zu `pinnedMessages` hinzugefügt.
- `deleteMessage()` löscht in beiden Listen eine zur `id` gehörenden `Message`, und gibt diese Message zurück. Falls die `Message` nicht existiert, wird `null` zurückgegeben und mit den `Listen` passiert logischerweise nichts.
- `editMessage()` soll den `content` der zur `id` passenden `Message` durch den übergebenen `String` ersetzen, falls die `Message` existiert. Bei Erfolg wird `true`, ansonsten `false` zurückgegeben.
- `toString()` gibt eine `String`-Repräsentation des `Topics` nach folgendem Schema zurück: `"Topic_' <Thread-Name> ':'\n <String-Repräsentation von messages>"`
- `pinnedToString()` gibt eine String-Repräsentation der gepinnten Nachrichten des Topics nach folgendem Schema zurück: `"Pinned_messages_in_Topic_' <Thread-Name> ':'\n <String-Repräsentation von pinnedMessages>"`

PinguTalk

In dieser Klasse wird nun letztendlich alles zusammengeführt. Ein neuer `PinguTalk` bekommt die Startlänge des `UserArrays` und die Größe des `Topic[]` in dieser Reihenfolge übergeben. Beide Arrays (`UserArray` und `Topic[]`) sollen dabei mit mindestens Größe `1` initialisiert werden.

- Auch hier sind wieder 2 statische Attribute `topicID` und `userID` zur eindeutigen Unterscheidung vorhanden. Sie benötigen ebenfalls keine Getter/Setter.
- `addMember()` erstellt aus der aktuellen `userID`, sowie den übergebenen Parametern `name` und `supervisor` ein neues `User`-Objekt, fügt das in das `UserArray` ein und gibt den neuen `User` zurück. Nach jedem neuen `User` soll die `userID` inkrementiert werden.
- `deleteMember()` löscht den zu einer `id` zugehörigen `User` im `UserArray` und gibt ihn zurück. Falls der `User` nicht existiert, soll `null` zurückgegeben werden.
- `createNewTopic()` erstellt ein neues `Topic` mit dem übergebenen `name`, fügt ihn an der ersten freien Stelle in `Topic[]` ein und gibt das neue `Topic` zurück. Falls kein Platz mehr frei ist, soll nichts passieren und `null` zurückgegeben werden. Nach jedem neuen eingefügten `Topic` soll die `topicID` inkrementiert werden.
- `deleteTopic()` löscht das zur `id` passenden `Topic` im Array und gibt dieses Topic zurück. Falls das `Topic` nicht vorhanden ist, soll `null` zurückgegeben werden.

Aufgabe

Implementiere die noch fehlenden Teile im Code!

? UserArray No results

In der Klasse `UserArray` sind die Methodenköpfe bereits vorhanden, allerdings fehlt noch deren Inhalt. Attribute, sowie Getter und Setter dieser Klasse fehlen auch noch. Implementiere `UserArray`, sodass die Klasse obiger Spezifikation entspricht.

? PinguTalk No results

In der Klasse `PinguTalk` fehlt noch alles. Implementiere auch hier alles, sodass die Klasse obiger Spezifikation entspricht.

? List-Funktionen No results

In der Klasse `List` sind die fünf Methoden `getByID()`, `megaMerge()`, `filterDays()`, `filterUsers()` und `toString()` noch nicht vorhanden. Implementiere auch diese!

[Lösungsvorschlag](#)

[Tests](#)



Exercise details

Release date:	Nov 17, 2022 18:30
Submission due:	Dec 4, 2022 18:00
Complaint due:	Dec 11, 2022 18:00

Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have **998** complaints left. 