Inlämningsuppgift P1 – Strömmar och trådar

2/2 Frågestund
7/2 Inlämning
8/2 Kamratgranskning
9/2 + 10/2 Redovisning

1 Inledning

Programmeringsuppgiften ska bidra till en grundläggande förståelse för:

- Användning av trådar, synkroniserad buffert och strömmar
- Implementering av Callback eller PropertyChangeListener

Uppgiften ska löses och redovisas individuellt och det krävs en individuell inlämning. Vi rekommenderar dock att diskutera uppgifterna och olika lösningsförslag i grupp. Det finns en frågestund den **2/2 kl 10:15** i samband med laborationen.

1.1 Klasser, interface och övriga filer som bifogas

Klasser: Buffer, ArrayProducer, MainP1, TestMessageProducer, TestMessageProducerInput inkl MPConsumer, TestProducer inkl MessageConsumer, Viewer

Interface: MessageProducer

Övriga filer: *new.txt* vilken ska placeras i katalogen *files* i projektet
new1.jpg, *new2.jpg*, ..., *new10.jpg* vilka ska placeras i katalogen *images* i projektet
javadoc-filerna ska placeras i mappen *docs* i projektet

1.2 Redovisning

Din lösning av uppgiften lämnas in via Canvas senast **kl 18 den 7/2**. Inlämningen ska innehålla samtliga klasser som används i lösningen. Klasserna Producer och TextfileProducer ska vara javadoc-kommenterade och javadoc ska vara genererad. Projektet namnges enligt samma regler som för Zip-filen.

Vid redovisningen den **9/2 och 10/2** kommer din lösning att köras med programmet MainP1.

Zip-filen ska du ge namnet AAABBB_P1.zip där AAA är de tre första bokstäverna i ditt efternamn och BBB är de tre första bokstäverna i ditt förnamn. Använd endast tecknen a-z när du namnger filen.

- Om Rolf Axelsson ska lämna in sina lösningar ska filen heta AxeRolP1.zip.
- Är ditt förnamn eller efternamn kortare än tre bokstäver så ta med de bokstäver som är i namnet: Janet Ek lämnar in filen EkJanP1.zip

1.3 Granskning

Senast **kl 10 den 8/2** kommer en kamrats lösning finnas i din inlämning på Canvas. Din uppgift är att även granska en kamrats lösningar på uppgifterna avseende:

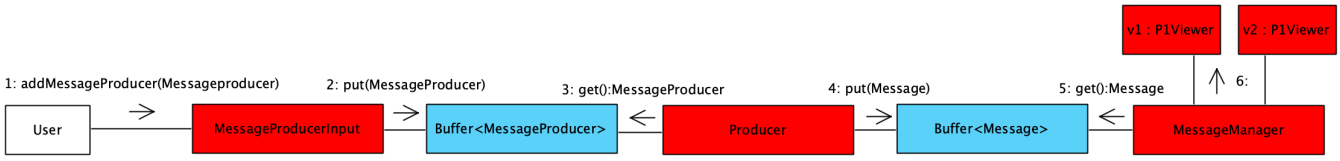
- Funktion – hur väl uppfyller lösningen kraven i uppgiften? Fungerar klasserna på avsett sätt?
- Kan du tänka dig något alternativt sätt att lösa uppgiften?
- Javadoc – är klasserna kommenterade enligt instruktion och är kommentarerna vettiga?

Resultatet av din granskning, 1-2 A4-sidor, ska du presentera i samband med redovisningen samt som en kommentar till kamratens inlämning.

2 Beskrivning av uppgiften

Applikationen är ett system, där en användare kan ladda upp meddelanden som sedan visas på olika skärmar (viewer). Tänk dig ett system med skärmar vid olika ställen i universitet (foajéer, bibliotek, …) där meddelanden ska visas (t.ex. nyheter, väder, bilder). För att underlätta processen finns det s.k. messageProducer, som genererar ett flertal meddelanden enligt ett visst schema (hur många gånger meddelandet ska visas, sekvenser av meddelanden, delay mellan två meddelanden).

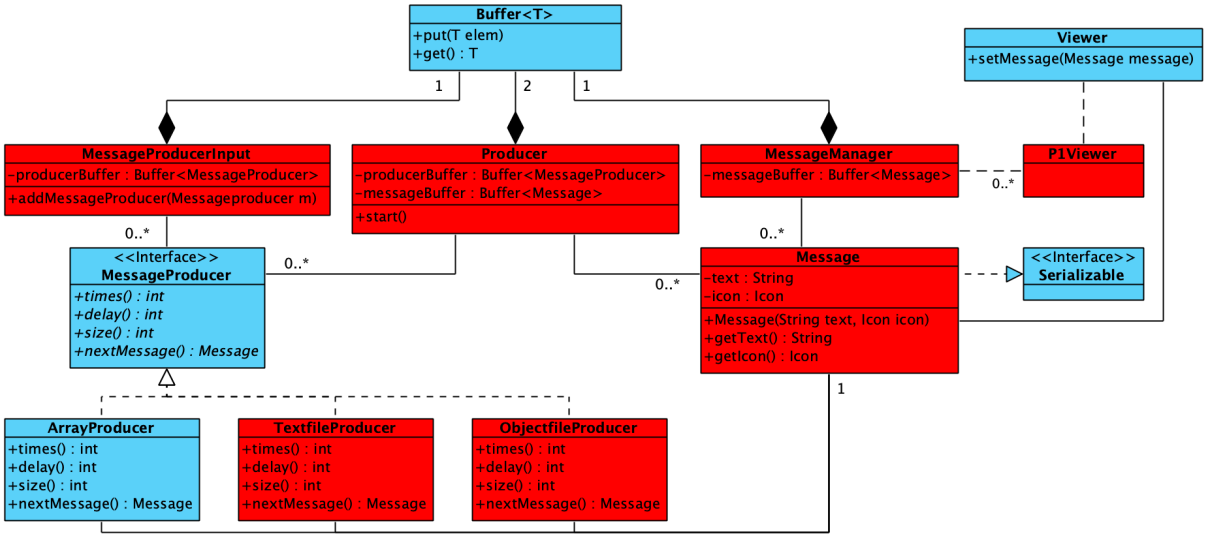
Figuren nedan visar några väsentliga klasser som ska ingå i din lösning. De rödmarkerade är klasser som du ska skriva. Det tillkommer några klasser att skriva i klassdiagrammet längre ner på sidan. I bilaga 1 finner du figuren i större storlek.



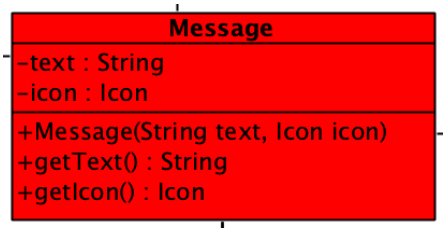
Figuren ovan visar hur kommunikationen i systemet ska ske.

1. Användare av systemet anropar metoden ***addMessageProducer*** i en *MessageProducerInput*-instans. Argument vid anropet är en MessageProducer-implementering, dvs en instans av en klass som implementertar MessageProducer.
2. **MessageProducerManager** placerar MessageProducer-implementeringen i en **Buffer<MessageProducer>**-instans, genom anrop till metoden *put*.
3. **Producer**, vilken använder en egen tråd, hämtar MessageProducer-implementering ur Buffer genom anrop till metoden *get*.
4. Varje hämtad MessageProducer-implementering ger ett antal **Message**-placeras i **Buffer<Message>**-instansen (anrop till metoden *put*)
5. **MessageManager** använder en egen tråd och hämtar Message-objekt ur Buffer<Message>-instansen.
6. Varje hämtat Message-objekt levereras till ett antal **P1Viewer**-objekt varvid Message-objektet visas i ett fönster.

Klassdiagrammet visar samtliga nödvändiga klasser i systemet. Steg för steg ska du skriva de rödfärgade klasserna.



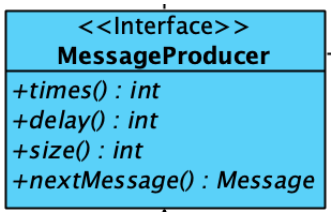
2.1 Message



Message är en klass vilken innehåller data och Message-objekt flödar genom systemet. Klassen *Message* har två instansvariabler, en lämplig konstruktor och get-metoder. Klassen ska implementera *Serializable* eftersom den används i strömmar (t.ex. i ObjectfileProducer).

Skriv klassen.

2.2 TextfileProducer

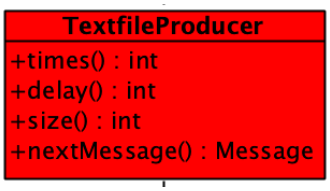


MessageProducer är ett interface som definierar funktionalitet för att

- hantera en sekvens av Message-objekt (*size()*, *nextMessage()*)
- ange hur lång tid varje Message-objekt ska visas (*delay()*, tid för samtliga Message-objekt)
- ange hur många gånger sekvensen ska visas (*times()*)

På så sätt går det bra att skapa ett bildspel (*delay* ett antal sekunder) eller en animation (*delay* delar av en sekund).

Klassen ***ArrayProducer*** ger exempel på en *MessageProducer*-implementering. Programmet *TestMessageProducer* visar ett exempel på vad man kan göra med en MessageProducer-implementering. Om du kör programmet kommer viewer visa en sekvens med text + bild.



Du ska skriva klassen **TextfileProducer**, vilken ska implementera *MessageProducer*. Studera klassen ArrayProducer innan du börjar!

Klassen ska ha konstruktorn:
public TextfileProducer(String filename)

Filen som är argument vid konstruktionen ska vara en textfil formatterad på följande sätt (se new.txt):

```
4
200
10
UP and
images/new1.jpg
 and...
images/new2.jpg
  and...
images/new3.jpg
   and...
images/new4.jpg
    and...
images/new5.jpg
     and...
images/new6.jpg
      DOWN and
images/new7.jpg
     and...
images/new8.jpg
   and...
images/new9.jpg
 and...
images/new10.jpg
```

Det första talet (4 ovan) är antalet gånger Message-sekvensen ska upprepas (times), det andra är tiden varje bild ska visas i millisekunder (200 ovan, delay), och det tredje talet är antalet par av text och bildfil som följer (10 par ovan, size). Av paren skapar du Message-objekt vilka lagras på lämpligt sätt i klassen.

Läs textfilen med en *BufferedReader*. Se till att ange teckenkodningen till "UTF-8". Det kan du göra om du använder in *InputStreamReader*.

I programmet TestMessageproducer kan du ersätta raden

```
    MessageProducer mp = getArrayProducer(4,500);
```

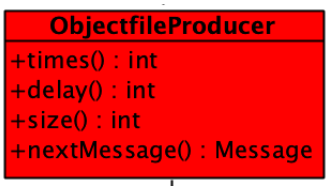
med

```
    MessageProducer mp = new TextfileProducer("files/new.txt");
```

Om du kör programmet får du ett körresultat liknande det tidigare men med texterna i *new.txt*. Bilderna är samma som i ArrayProducern som användes.

2.3 ObjectfileProducer

ObjectfileProducer ska implementera MessageProducer. Klassen läser, precis som TextfileProducer, från en fil. Denna fil ska vara formaterad på följande sätt:



Antal repetitioner (times), Fördöjning (delay), Antal Message-objekt (size), size stycken Message-objekt

t.ex.: 5, 10000, 3, Message-objekt, Message-objekt, Message-objekt

När du är färdig med klassen kan du, i programmet TestMessageProducer, ersätta raden

```
MessageProducer mp = new TextfileProducer("files/new.txt");
```

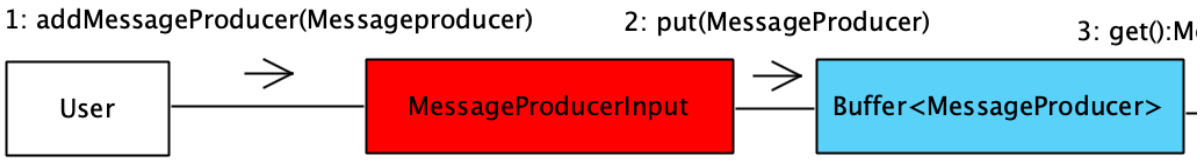
med raderna

```
writeToObjectStream("files/new.dat",getArrayProducer(4,500));
MessageProducer mp = new ObjectfileProducer("files/new.dat");
```

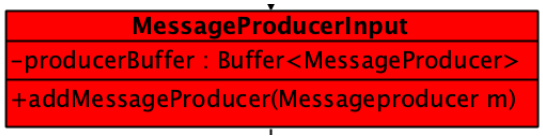
Om du kör programmet får du ett körresultat identiskt med det då ArrayProducer användes.

2.4 MessageProducerInput

MessageProducerInput-objektet får vid anrop till metoden *addMessageProducer (1)* tillgång till en instans av en klass vilken implementerar *MessageProducer*. Detta objekt ska placeras i en buffert av typen ***Buffer<MessageProducer>*** med hjälp av *put*-metoden (2).



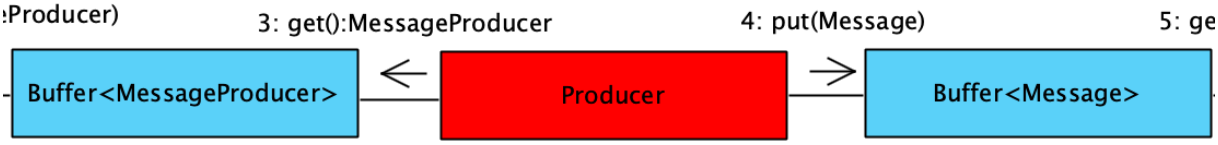
Skriv klassen **MessageProducerInput**. I klassdiagrammet ser du instansvariabler och metoder som *MessageProducerInput* måste innehålla. Klassen ska dessutom innehålla konstruktorn



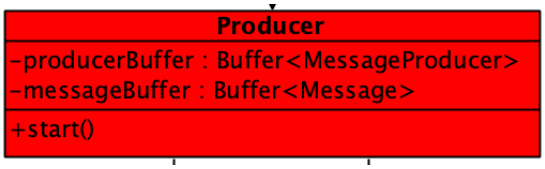

```
public MessageProducerInput(Buffer<Messageproducer>)
```

Med programmet ***TestMessageProducerInput*** kan du testa *MessageProducerInput* och att *MessageProducer*-implementeringar placeras i bufferten. ***MPConsumer*** ger exempel på hur en tråd hämtar *MessageProducer*-implementeringar ur *Buffer*-instansen. *MPConsumer*-klassen är i filen *TestMessageProducerInput.java*.

2.5 Producer



Producer-instansen ska fungera så här:



3. Hämta en *MessageProducer*-implementering ur bufferten till vänster i figuren (*get()*).
4. Använd *MessageProducer*-implementeringen för att placera *Message*-instanser i bufferten till höger i figuren (*put()*). Men de ska inte placeras i bufferten så fort det går utan med en viss paus mellan varje objekt. Det är metoden *delay* i *MessageProducer*-implementeringen som ger pausen. Metoden *times* anger hur många gånger *Message*-sekvensen ska placeras i bufferten. Metoden *size* anger hur många *Message*-objekt det är i sekvensen. Och slutligen returnerar metoden *nextMessge* *Message*-instanser, en i taget. När sekvensen är slut så returneras det första elementet på nytt.

Därefter upprepas 3 och 4 på nytt.

Din uppgift är att skriva klassen ***Producer***. Klassen ska använda en tråd och konstruktorn

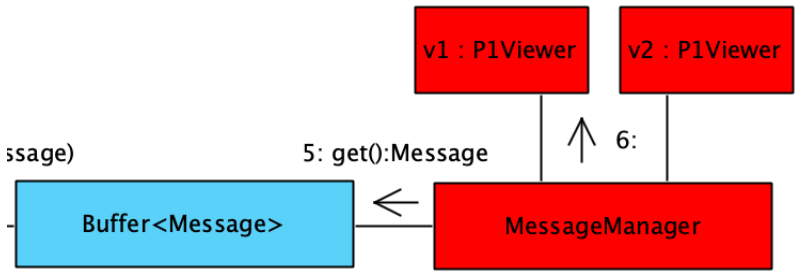
```
public Producer(Buffer<MessageProducer prodBuffer,
                Buffer<Message> messageBuffer)
```

Slutligen ska man kunna starta tråden genom att anropa metoden *start*.

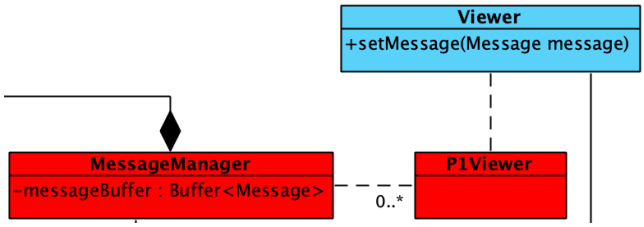
Programmet ***TestProducer*** (vilket förutsätter att *TestMessageProducerInput* fungerar) visar resultatet i en *Viewer*. Klassen ***MessageConsumer*** ger exempel på hur man hämtar *Message*-objekt ur Buffer<Message>-instansen. *MessageConsumer* är i filen *TestProducer.java*.

MPConsumer ger exempel på hur en tråd hämtar *MessageProducer*-implementeringar ur *Buffer*-instansen. *MPConsumer*-klassen är i filen *TestMessageProducerInput.java*.

2.6 MessageManager och P1Viewer



I denna del av systemet ska ***MessageManager***-objektet hämta *Message*-instanser ur bufferten och se till att dessa visas i ett antal ***P1Viewer***-instanser. Denna överföring ska du ordna genom att antingen använda genom att definiera ett interface för ***Callback*** eller genom att implementera en ***PropertyChangeListener***.



Som du ser i ***MainP1*** så känner inte *MessageManager*-objektet till *P1Viewer*-objekten efter instansiering. Däremot känner *P1Viewer*-objekten till *MessageManager*-instansen:

```
P1Viewer viewer = new P1Viewer(messageManager,640,480);
```

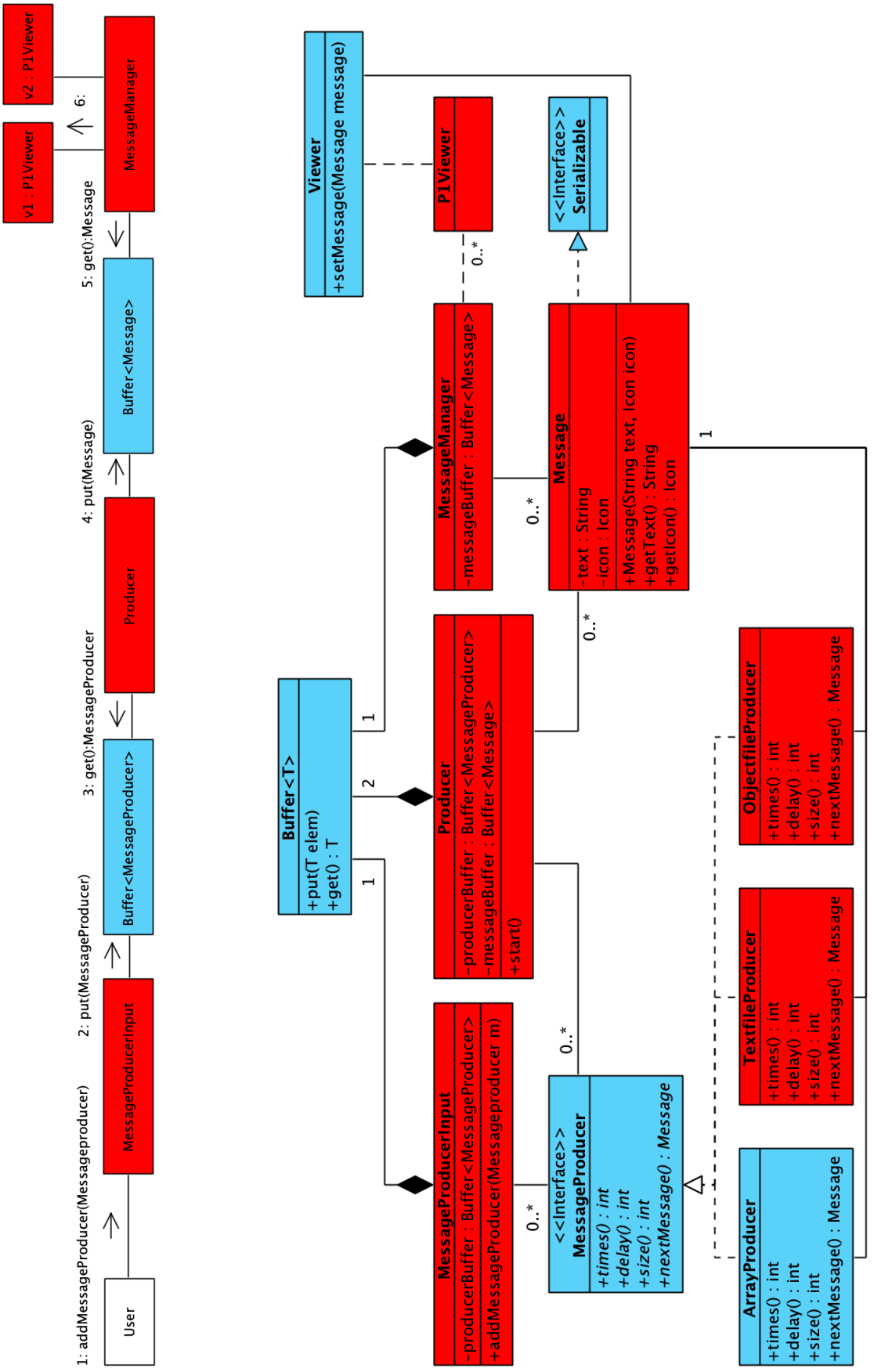
Och kan därför registrera någon form av lyssnare. När du skriver *P1Viewer* ska du använda dig av *Viewer*-klassen. *P1Viewer* ska antingen ha en *Viewer* eller ärva en *Viewer*. Avgör själv vilket som är lämpligast. Oavsett vilket du väljer måster *P1Viewer* ha en konstruktor som tar tre argument: en *MessageManager*-instans, width och height.

Denna dels av system får du testa på egen hand. Till det levereras inget program.

Obs! Om du låter P1Viewer ha en Viewer som attribut så kan klassen innehåll en getViewer-metod. När viewern ska visas så används metoden (i MainP1):

```
Viewer.showPanelInFrame(v1.getViewer(), "Viewer 1", 100, 50);
```

Bilaga 1


1: addMessageProducer(Messageproducer)
2: put(MessageProducer)
3: get():MessageProducer
4: put(Message)
5: get():Message
6:
User
MessageProducerInput
Buffer<MessageProducer>
Producer
Buffer<Message>
MessageManager
v1 : P1Viewer
v2 : P1Viewer
Viewer
+setMessage(Message message)
P1Viewer
<<Interface>>
Serializable
Buffer<T>
+put(T elem)
+get() : T
MessageManager
-messageBuffer : Buffer<Message>
Message
-text : String
-icon : Icon
+Message(String text, Icon icon)
+getText() : String
+getIcon() : Icon
Producer
-producerBuffer : Buffer<MessageProducer>
-messageBuffer : Buffer<Message>
+start()
MessageProducerInput
-producerBuffer : Buffer<MessageProducer>
+addMessageProducer(MessageProducer m)
<<Interface>>
MessageProducer
+times() : int
+delay() : int
+size() : int
+nextMessage() : Message
ObjectfileProducer
+times() : int
+delay() : int
+size() : int
+nextMessage() : Message
TextfileProducer
+times() : int
+delay() : int
+size() : int
+nextMessage() : Message
ArrayProducer
+times() : int
+delay() : int
+size() : int
+nextMessage() : Message
0..*
1
2