7.3.6 Klasse Circular (gb.data)

Die Klasse *Circular* präsentiert einen zirkulären Puffer als Speicherbereich mit fester Größe. Wenn der Puffer voll ist, werden die ältesten Elemente im Puffer mit neuen Elementen nur dann überschrieben, wenn die Eigenschaft *Circular.Overwrite = True* gesetzt wurde. Alle Elemente eines Circular-Objekts sind vom Typ Variant.

Ein Circular hat einen konstanten Speicherbedarf, denn es ist intern ein Array mit fixer Anzahl von Elementen. Ein Circular mit beispielsweise acht Elementen wird niemals mehr Speicher verbrauchen als *8 * SizeOf(gb.Variant)*.

Zusätzlich verfügt ein Circular über einen Lese- und Schreib-Zeiger. Diese beiden Eigenschaften (.Reader und .Writer) sind getrennt, um eine Unterscheidung von Daten-Produzent (die Write()-Methode) und Daten-Konsument (Read()-Methode) zu gewährleisten, welche asynchron dasselbe Objekt manipulieren können.

Erreicht der Lese-Zeiger den Schreib-Zeiger, so existieren keine weiteren ungelesenen Elemente und der Circular wird als "leer" bezeichnet. Erreicht hingegen der Schreib-Zeiger den Lese-Zeiger, so ist kein Platz mehr im Circular für neue Elemente und er wird als "voll" deklariert.

7.3.6.1 Eigenschaften

Die Eigenschaften des Circular:

Eigenschaft	Datentyp	Beschreibung
.Size	Integer	Lesen oder Setzen der Anzahl der Elemente (→ Methode Resize())
.IsEmpty	Boolean (ReadOnly)	Wahr, wenn keine Elemente im Circular sind, sonst falsch
.IsFull	Boolean (ReadOnly)	Wahr, wenn alle Plätze im Circular besetzt sind, sonst falsch
.Overwrite	Boolean	Wenn wahr, so wird das älteste Element überschrieben, wenn der Circular voll ist, ansonsten schlägt jeder weitere Schreibversuch in einen vollen Circular fehl (ohne einen Fehler zu produzieren).
.Reader	Integer	Index des Lese-Zeigers
.Writer	Integer	Index des Schreib-Zeigers

Tabelle 7.3.6.1.1: Eigenschaften der Klasse Circular

7.3.6.2 Methoden eines Circular-Objektes

Methode	Beschreibung
Clear()	Entfernen aller Elemente aus dem Puffer
Read()	Lesen des ältesten Elements oder Null, wenn der Circular leer ist
Peek()	Read() ohne den Lese-Zeiger weiter zu rücken
Write(vElement)	Schreiben eines neuen Elements. Wenn der Circular voll ist, hängt die Auswirkung der Methode vom Wert der Eigenschaft *Overwrite* ab.
Reset()	Lese- *und* Schreib-Zeiger auf Index 0 zurücksetzen
Resize(iSize)	Ändern der Größe des Circular

Tabelle 7.3.6.2.1: Methoden der Klasse Circular

Achtung:
Die Methode *Circular.Resize()* belässt die Relation der Eigenschaften *Reader* und *Writer* in einem undefinierten Zustand.

Es wird deshalb dringend empfohlen, jedem Resize() ein Reset() folgen zu lassen, um zufällige Abstürze infolge falscher Annahmen zu vermeiden. Alternativ können auch beide Zeiger denselben Wert besitzen, wenn Resize() aufgerufen wird. In diesem Fall haben beide Zeiger nach der Größenänderung des Circulars ebenfalls denselben Wert.

7.3.6.3 Projekt zum Einsatz der Klasse Circular

Durch den Einsatz eines zirkulären Puffers lässt sich die maximale Speichernutzung einer Anwendung auf eine konstante Größe reduzieren, da alte Daten automatisch mit neuen überschrieben werden. Dieser Mechanismus kommt bei *Programm-Logs* zum Einsatz, da diese vornehmlich dann interessant werden, wenn ein Programm Fehler produziert. Die unmittelbar *vor dem Absturz* des Programms geschriebenen Log-Informationen können auf den Programmteil verweisen, der den Fehler erzeugte. Diese typische Anwendung *Programm-Log* mit zirkulären Puffern wird im angebotenen Projekt demonstriert.

Um die Lösung offen zu halten wird ein Client-Server-System verwendet, das einen netzwerkfähigen System-Log-Services implementiert, der für jedes angeschlossene Programm (Client) ein eigenes Log führt.

- Nur der Server legt für jedes (Client-)Programm ein Log an und pflegt den Inhalt der einzelnen Logs.
- Jedes Log setzt die Klasse *Circular* ein. Es kann festgelegt werden, wie viele Elemente ein solcher zirkulärer Puffer hat.
- Die Anzahl der Elemente bestimmt die Größe des zirkulären Puffers.

- Auf das Log kann nur zugegriffen werden, wenn das (Client-)Programm regulär beendet wurde oder abgestürzt ist.

In der folgenden Beschreibung wird davon ausgegangen, dass 2 (Client-)Programme mit dem Server verbunden sind:

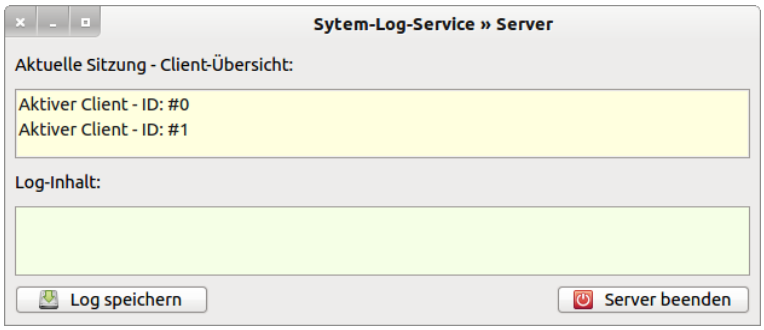

Abbildung 7.3.6.3.1: Zwei aktive (Client-)Programme sind am Server angemeldet

Der Server weist dem ersten (Client-)Programm eine ID zu und legt eine (temporäre) Log-Datei an:

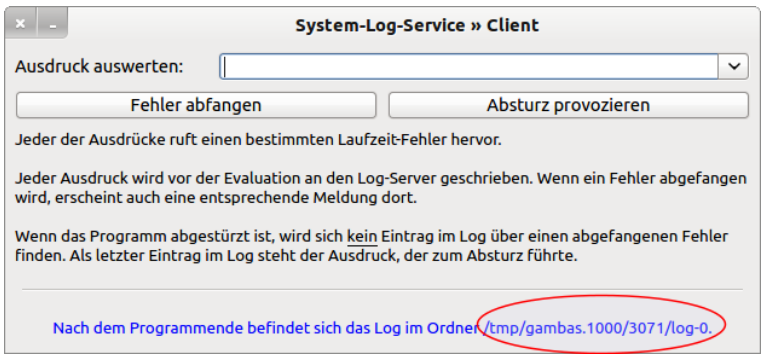

Abbildung 7.3.6.3.2: (Client-)Programm 0

Sie können nun mit dem Programm 0 arbeiten, indem Sie unterschiedliche Ausdrücke aus der Combo-Box auswählen oder eigene Ausdrücke dort eintragen. Sie können einen Fehler abfangen oder einen Programm-Absturz provozieren. In diesem Fehler-Fall wird das Programm 0 beendet.

In der nächsten Abbildung 7.3.6.3.3 erkennen Sie, dass in der Client-Übersicht eine entsprechende Meldung generiert wurde. Außerdem wird angezeigt, dass Sie nicht auf das Log des 2. *aktiven* (Client-)Programms zugreifen können – der Zugriff wird verweigert!

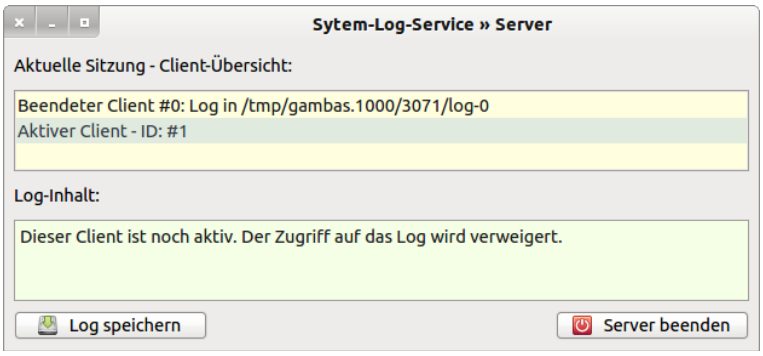

Abbildung 7.3.6.3.3: Server mit Eintragungen und Warnungen

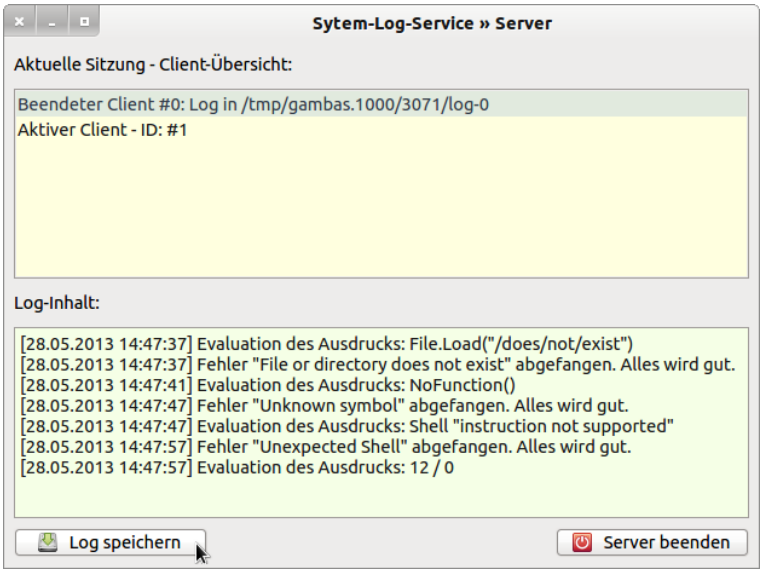

Abbildung 7.3.6.3.4: Ansicht des Inhalts des Logs von Client 0

Der Inhalt eines Logs wird nur dann angezeigt, wenn man auf den Eintrag "Beendeter Client #.." klickt.

Da jedes Log nur zur Laufzeit des Servers existiert, kann man das Log für jedes (Client-)Programm sichern und frei abspeichern.

Ergänzung:
Die neue Komponente *gb.logging* vom Sebastian Kulesz stellt eine flexible API für das Logging und Nachverfolgen von Ereignissen bereit, während das Programm läuft.