# Reference Counting

**Seminar „Effiziente Programmierung in C“**

Moritz Gaack

05. Dezember 2013

# Gliederung

# Motivation

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
    int * p1;
    int * p2 = (int *)calloc(1, sizeof(int));
    int * p3 = (int *)calloc(1, sizeof(int));

    p3 = p2;   // Original p3 has been lost.

    free(p1);  // Memory corruption.
    free(p2);  // p3 is invalid now.
    free(p3);  // Memory corruption.

    return 0;
}
```

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

1. **Freigeben** oder **Überschreiben** von Speicherblöcken, die noch **in Verwendung** sind

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

1. **Freigeben** oder **Überschreiben** von Speicherblöcken, die noch **in Verwendung** sind $\rightarrow$ **Memory Corruption**

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

1. **Freigeben** oder **Überschreiben** von Speicherblöcken, die noch **in Verwendung** sind $\rightarrow$ **Memory Corruption**:
   - Absturz der Applikation
   - Korrupte Benutzerdaten

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

1. **Freigeben** oder **Überschreiben** von Speicherblöcken, die noch **in Verwendung** sind $\rightarrow$ **Memory Corruption**:
   - Absturz der Applikation
   - Korrupte Benutzerdaten
2. **Nicht freigegebene** Speicherblöcke, die **nicht mehr in Gebrauch** sind

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

1. **Freigeben** oder **Überschreiben** von Speicherblöcken, die noch **in Verwendung** sind $\rightarrow$ **Memory Corruption**:
   - Absturz der Applikation
   - Korrupte Benutzerdaten
2. **Nicht freigegebene** Speicherblöcke, die **nicht mehr in Gebrauch** sind $\rightarrow$ **Memory Leaks**

## Allgemeine Problemstellung

**Inkorrektes Speichermanagement**[3]:

1. **Freigeben** oder **Überschreiben** von Speicherblöcken, die noch **in Verwendung** sind $\rightarrow$ **Memory Corruption**:
   - Absturz der Applikation
   - Korrupte Benutzerdaten
2. **Nicht freigegebene** Speicherblöcke, die **nicht mehr in Gebrauch** sind $\rightarrow$ **Memory Leaks**:
   - Nicht befreite Speicherbereiche sind blockiert
   - Applikation benötigt immer mehr Speicher
     - Schlechte Systemperformance
     - Beenden der Anwendung

# Klassische Konzepte für Speichermanagement

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*
   2. Nicht-konservative automatische Speicherbereinigung

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*
   2. Nicht-konservative automatische Speicherbereinigung
      - Tracing garbage collectors
      - **Reference Counting** (*MRR*, *ARC*)

[1] http://de.wikipedia.org/wiki/Garbage_Collection
[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*
   2. Nicht-konservative automatische Speicherbereinigung
      - Tracing garbage collectors
      - **Reference Counting** (*MRR*, *ARC*)

- Wann wird der Speicher freigegeben?

[1] `http://de.wikipedia.org/wiki/Garbage_Collection`
[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*
   2. Nicht-konservative automatische Speicherbereinigung
      - Tracing garbage collectors
      - **Reference Counting** (*MRR*, *ARC*)

   ▶ Wann wird der Speicher freigegeben?
   *Implementierungsabhängig.*

[1] `http://de.wikipedia.org/wiki/Garbage_Collection`
[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*
   2. Nicht-konservative automatische Speicherbereinigung
      - Tracing garbage collectors
      - **Reference Counting** (*MRR*, *ARC*)

   - Wann wird der Speicher freigegeben?
     *Implementierungsabhängig.*
   - C-Bibliotheken: z. B. *Boehm GC*[2]

[1] `http://de.wikipedia.org/wiki/Garbage_Collection`
[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

## Klassische Konzepte für Speichermanagement

1. **Eigenes Speichermanagement** mit Standard C 
2. **Garbage Collection**[1]
   1. Konservative automatische Speicherbereinigung
      *Keine zuverlässige Erkennung von nicht-referenzierten Objekten.*
   2. Nicht-konservative automatische Speicherbereinigung
      - Tracing garbage collectors
      - **Reference Counting** (*MRR*, *ARC*)

   - Wann wird der Speicher freigegeben?
     *Implementierungsabhängig.*
   - C-Bibliotheken: z. B. *Boehm GC*[2]
   - Effizient?

[1] `http://de.wikipedia.org/wiki/Garbage_Collection`
[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

# Übersicht Reference Counting[3]

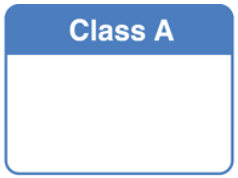


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

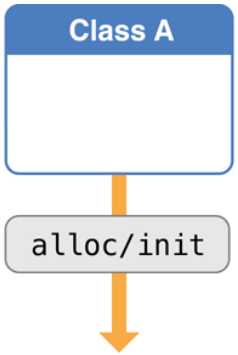


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

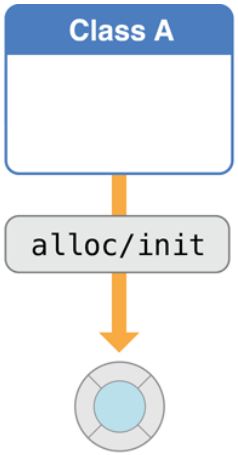


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

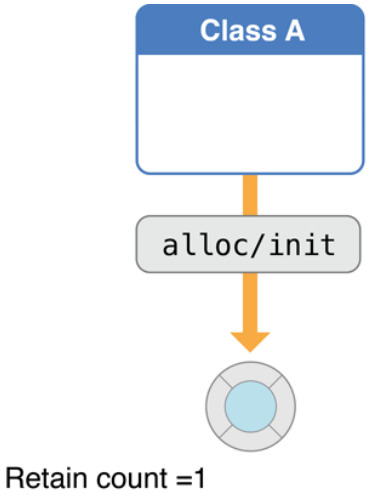


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

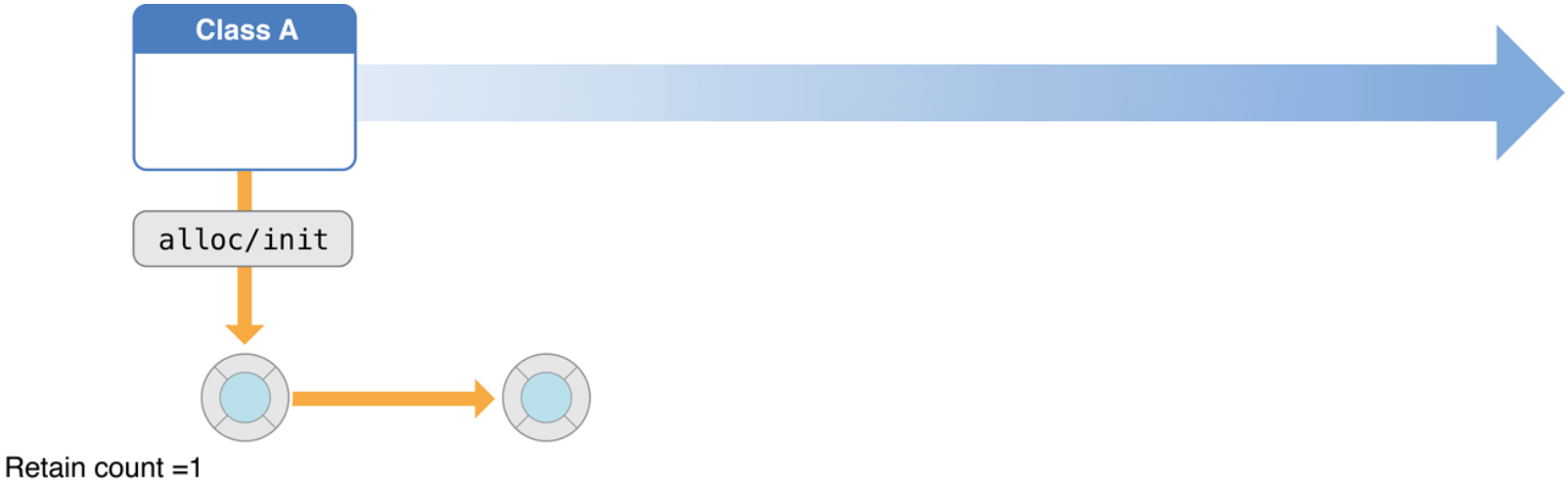


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

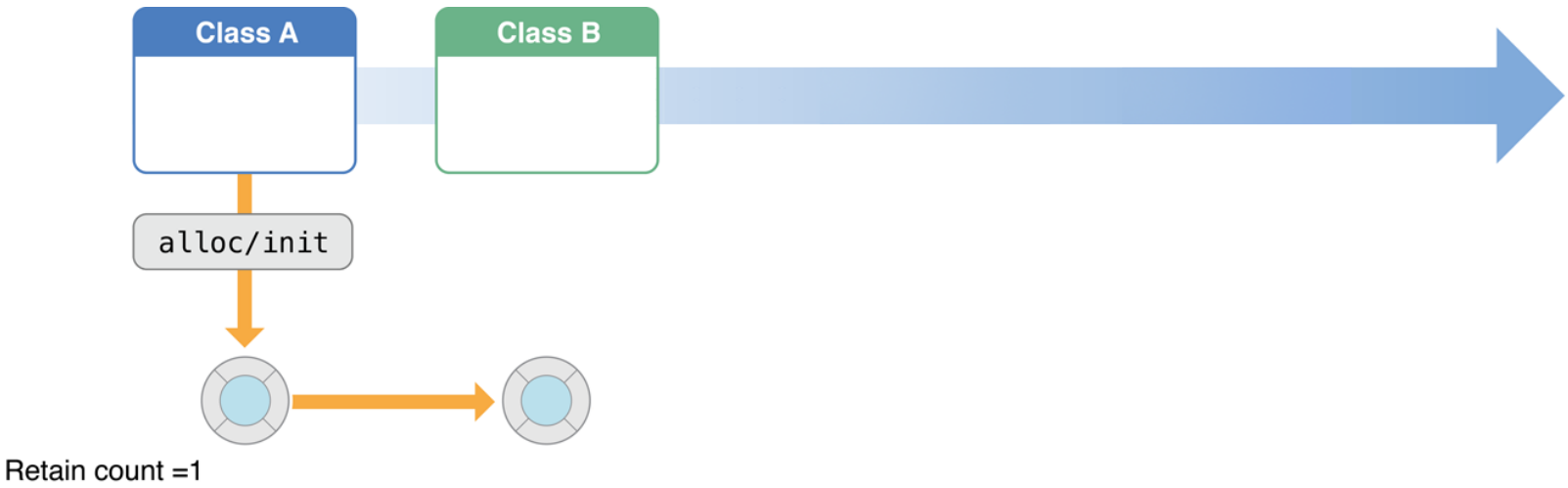


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

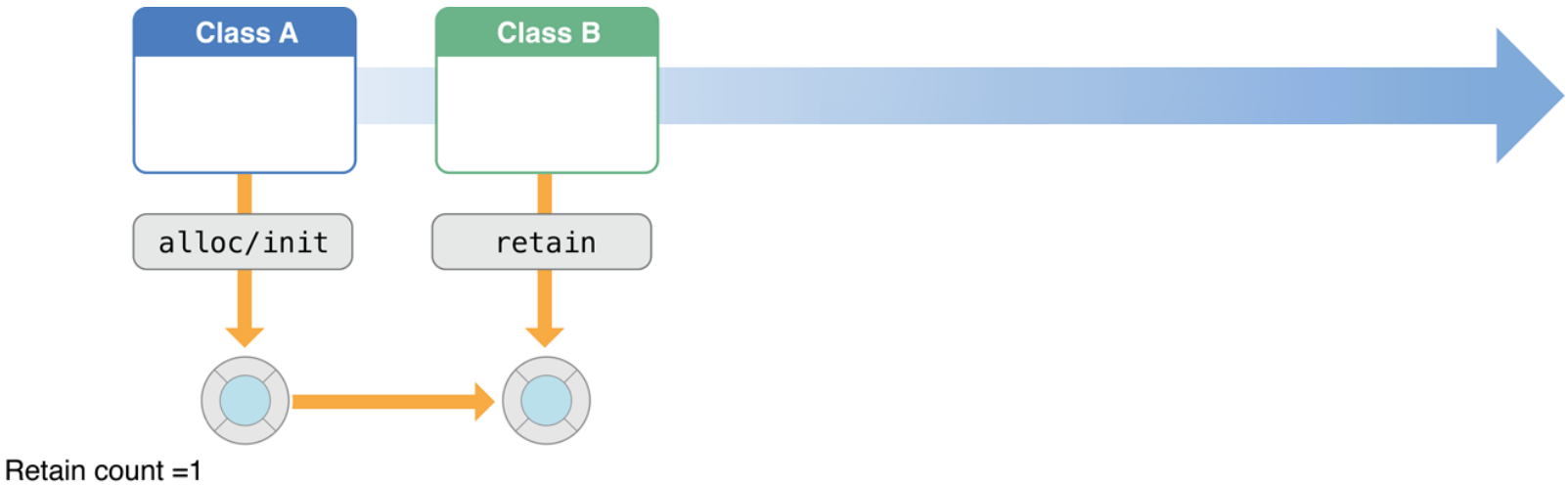


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

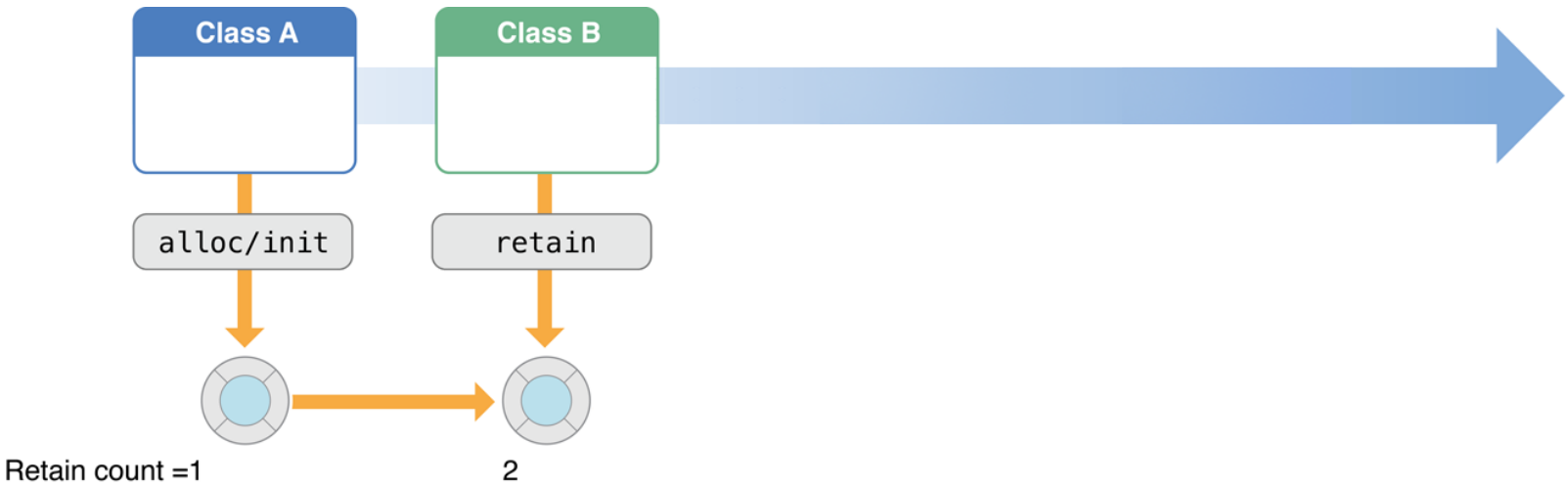


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

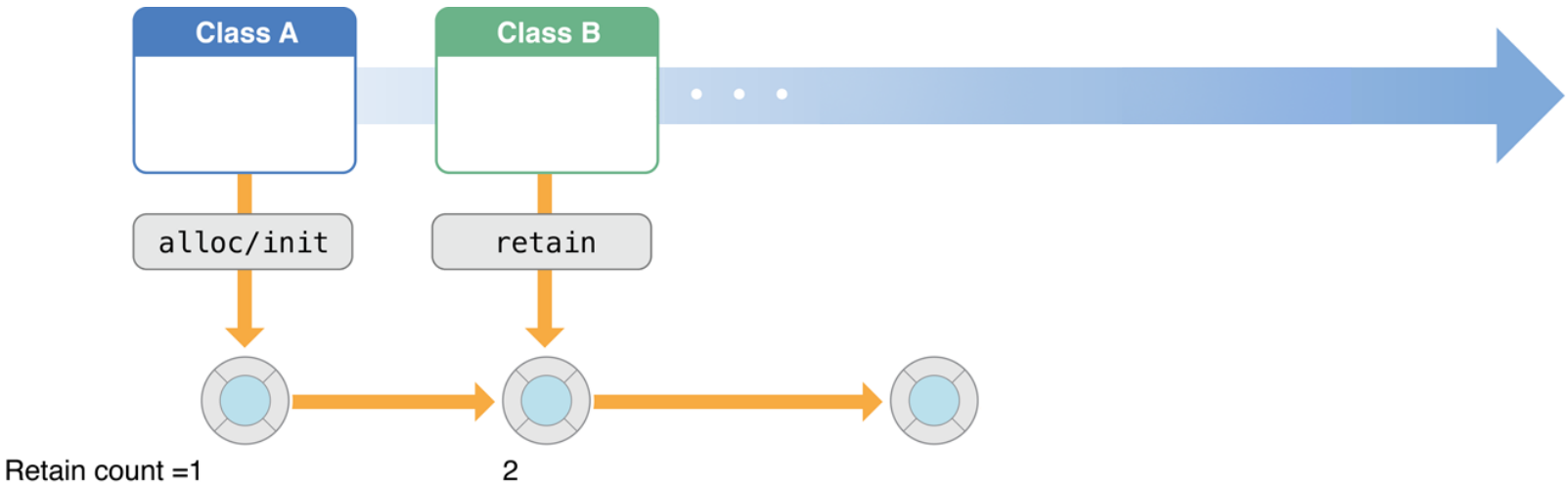


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

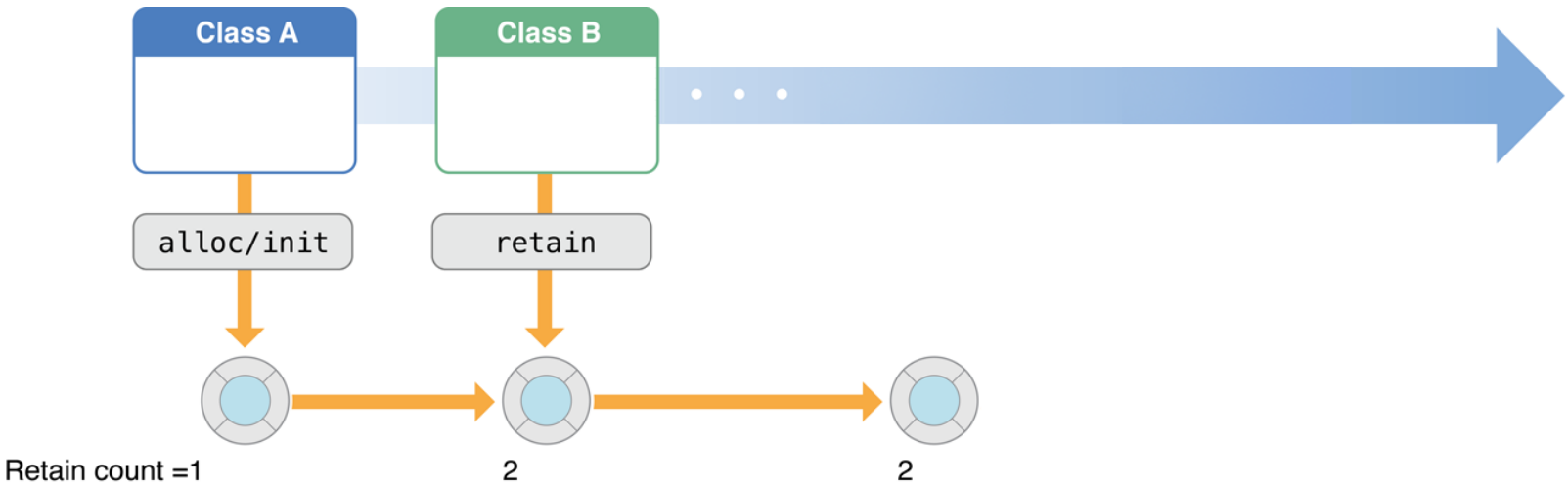


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

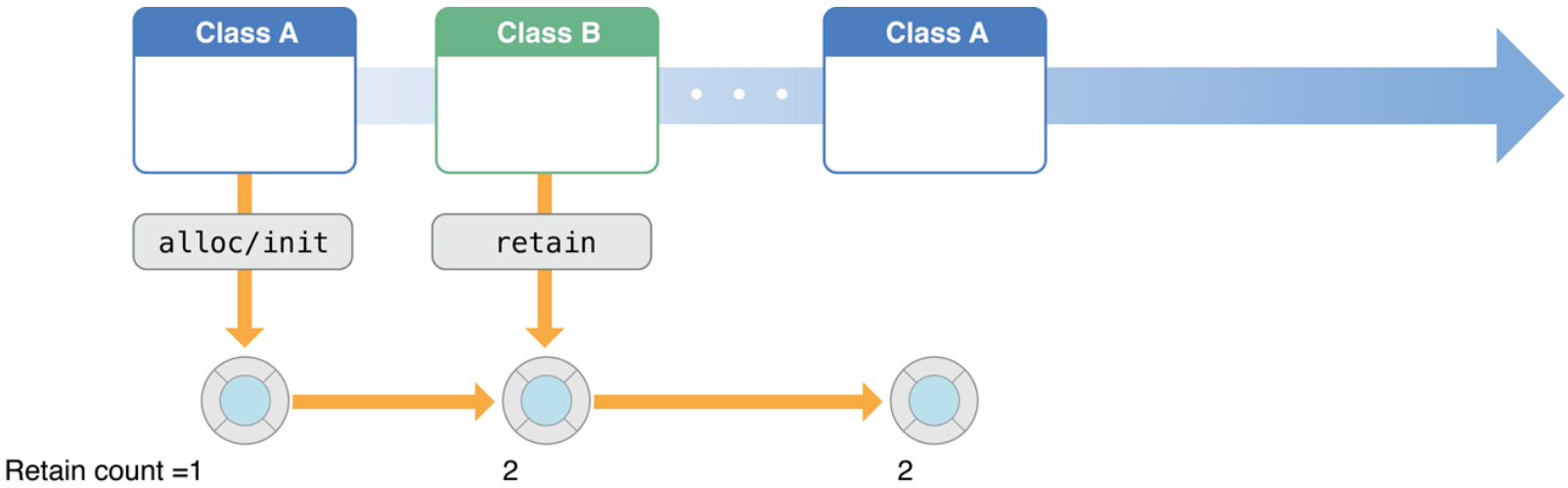


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

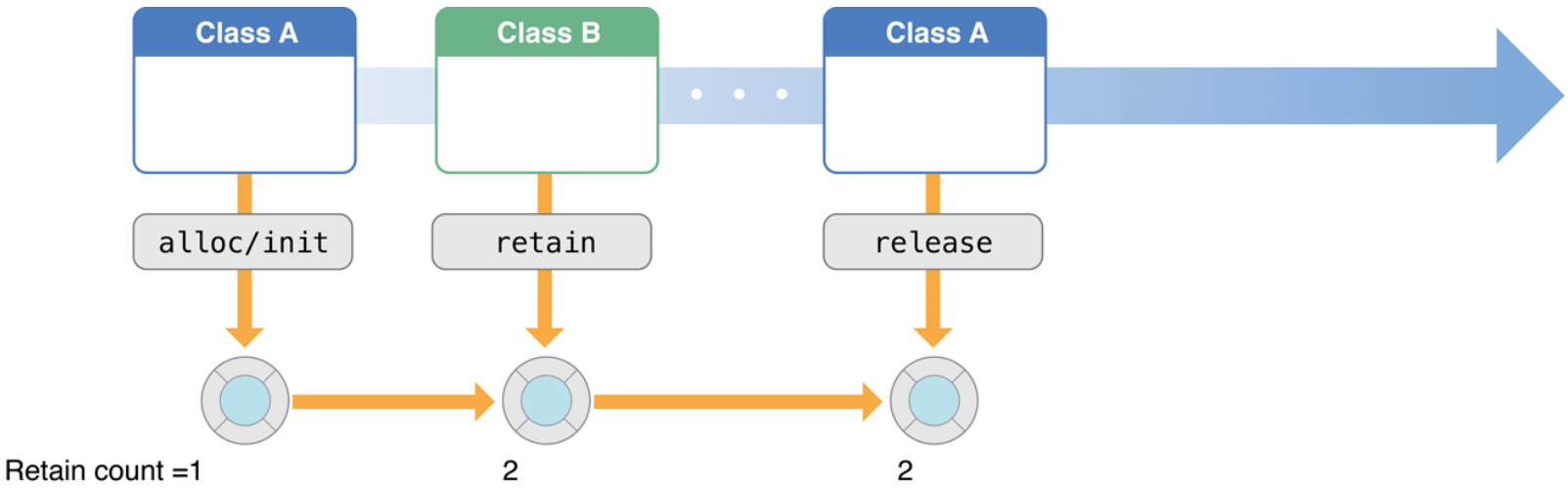


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

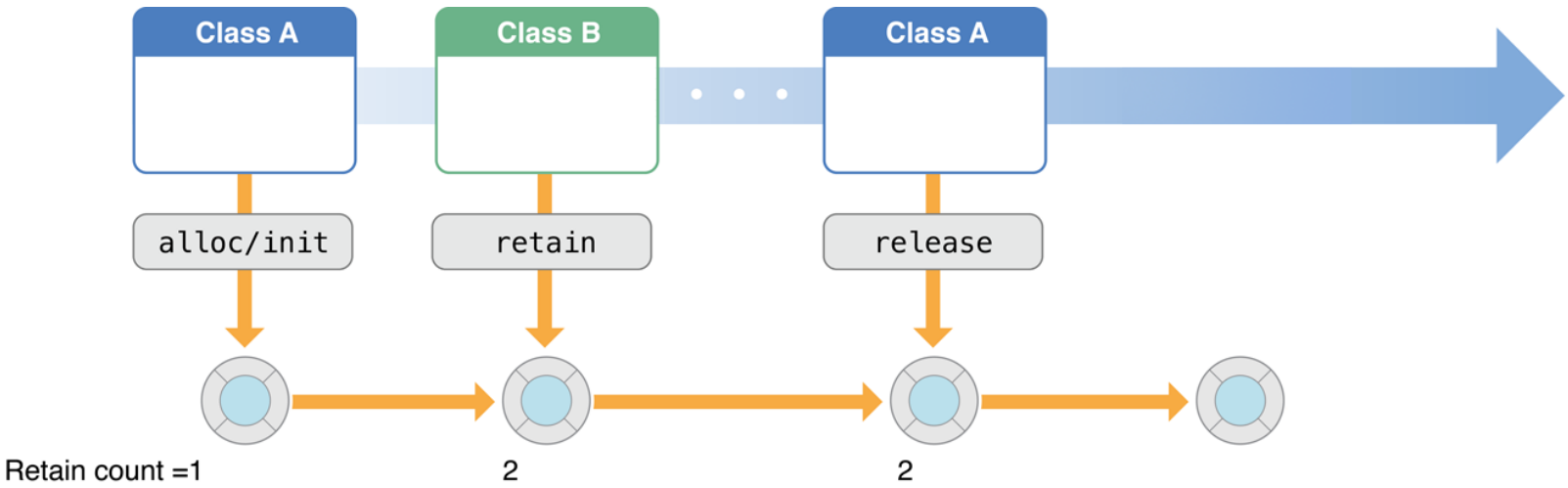


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

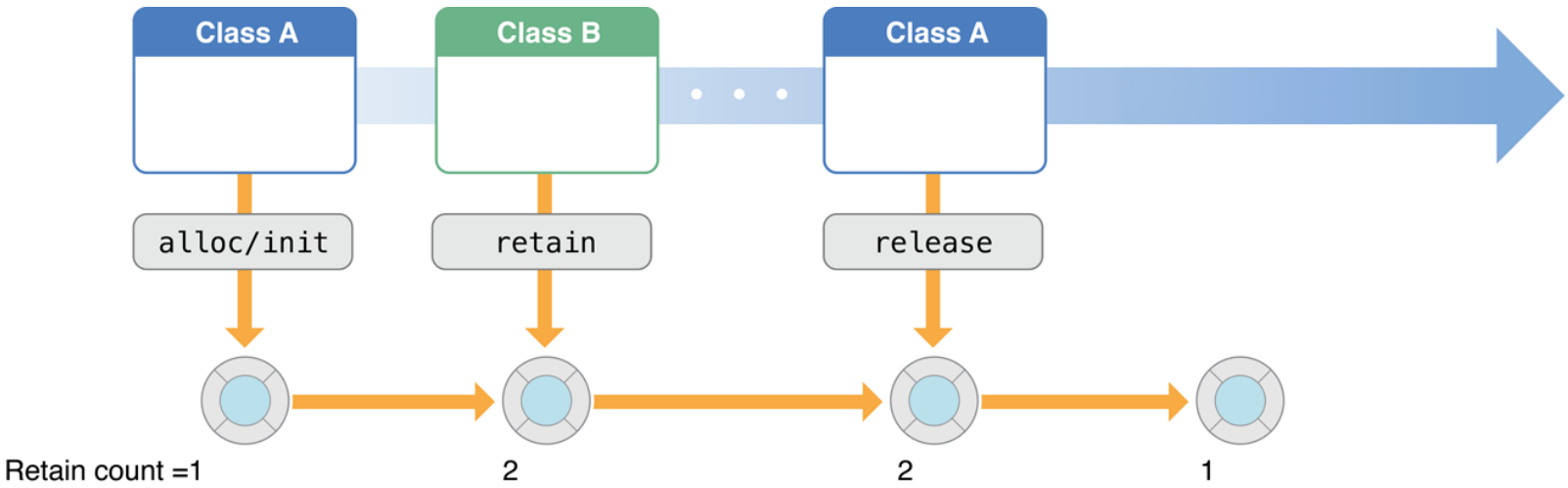


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

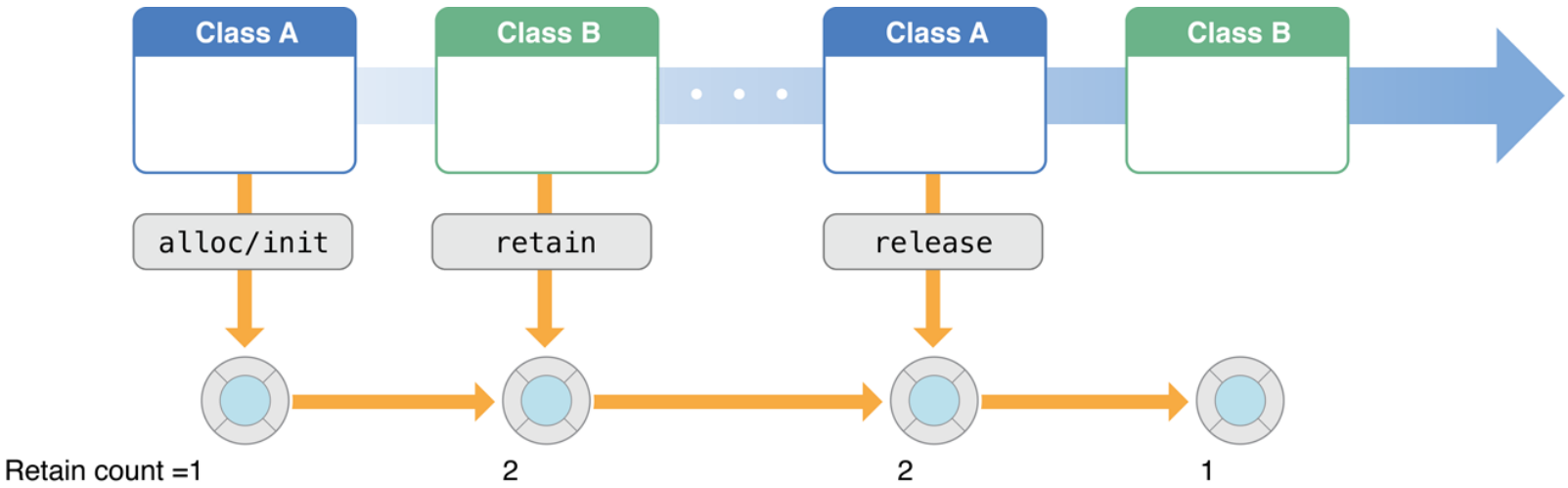


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

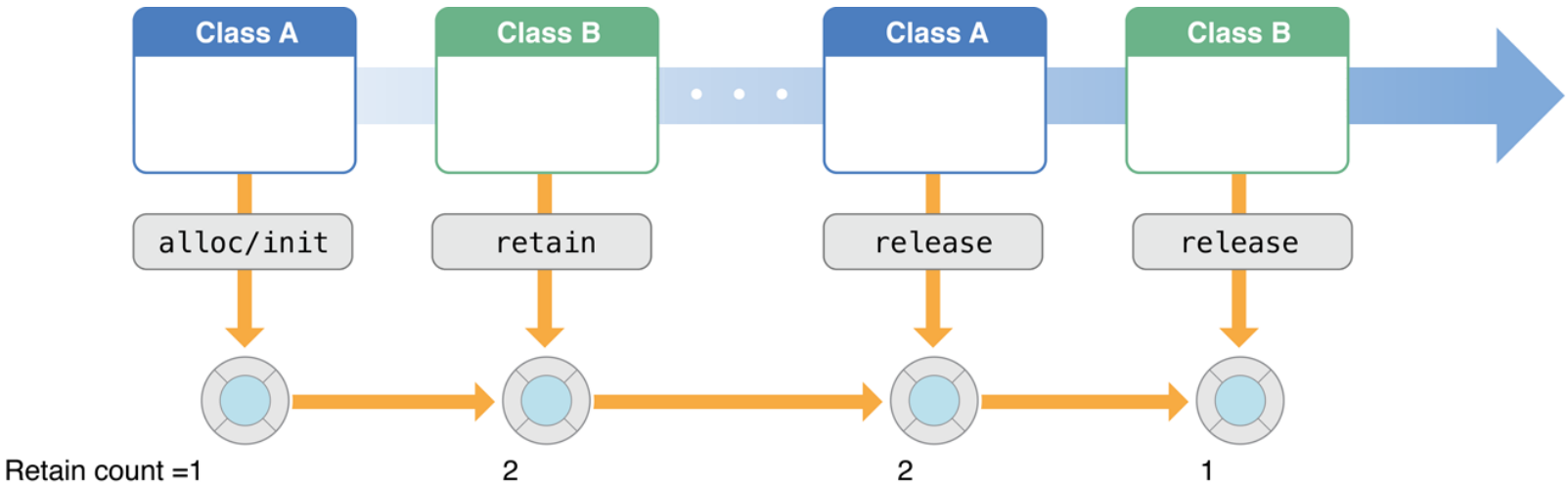


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

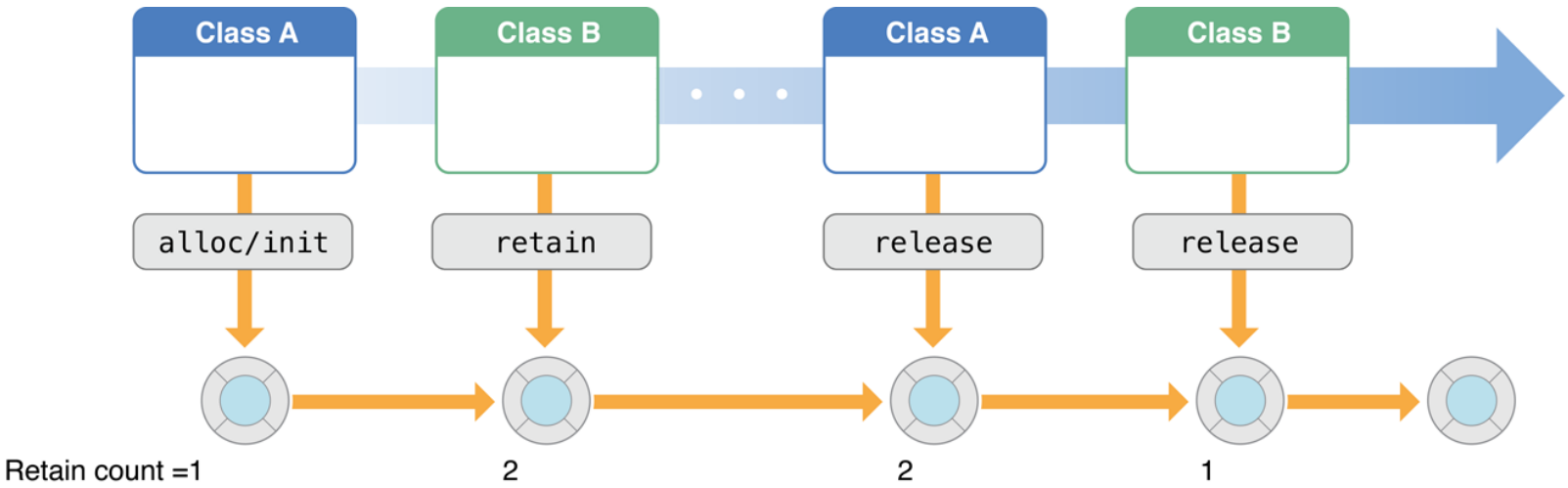


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

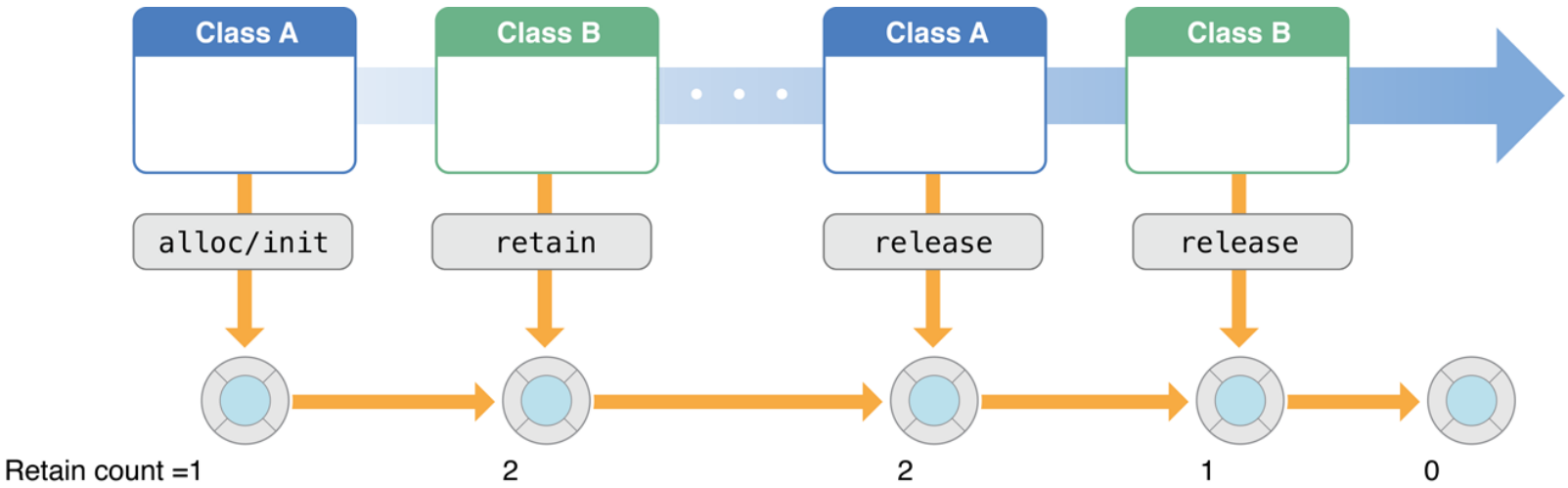


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

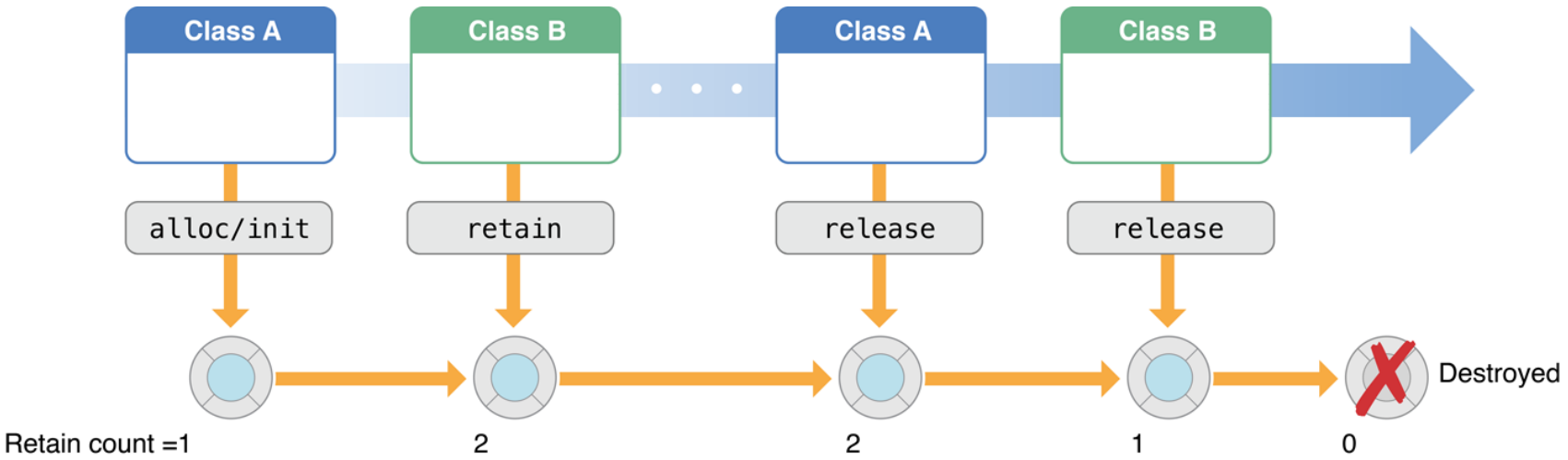


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

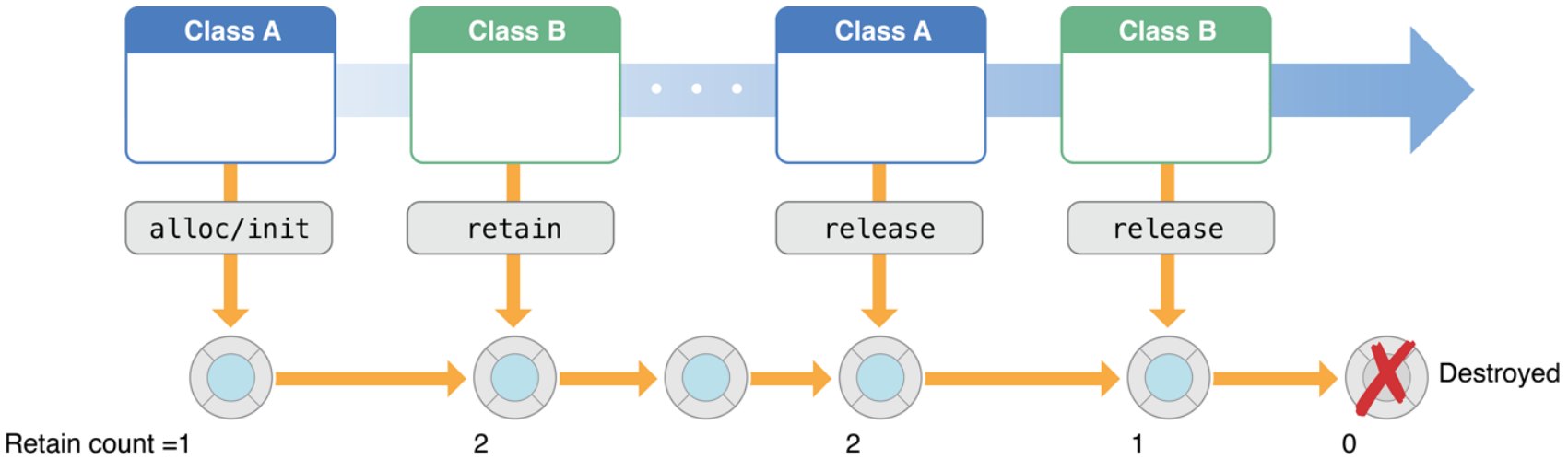


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

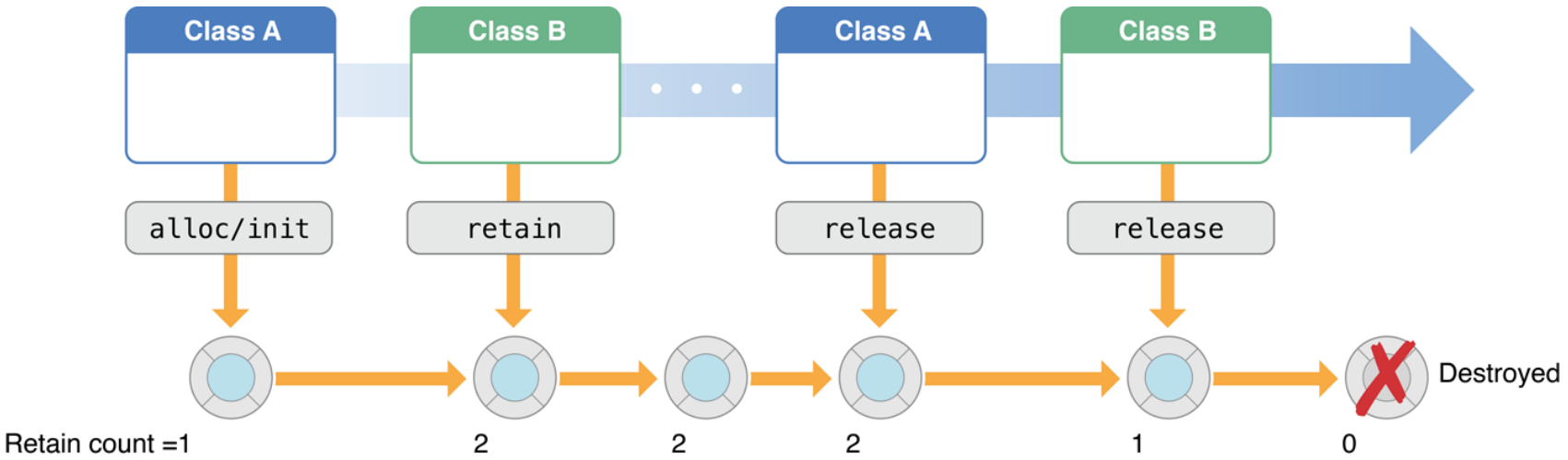


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

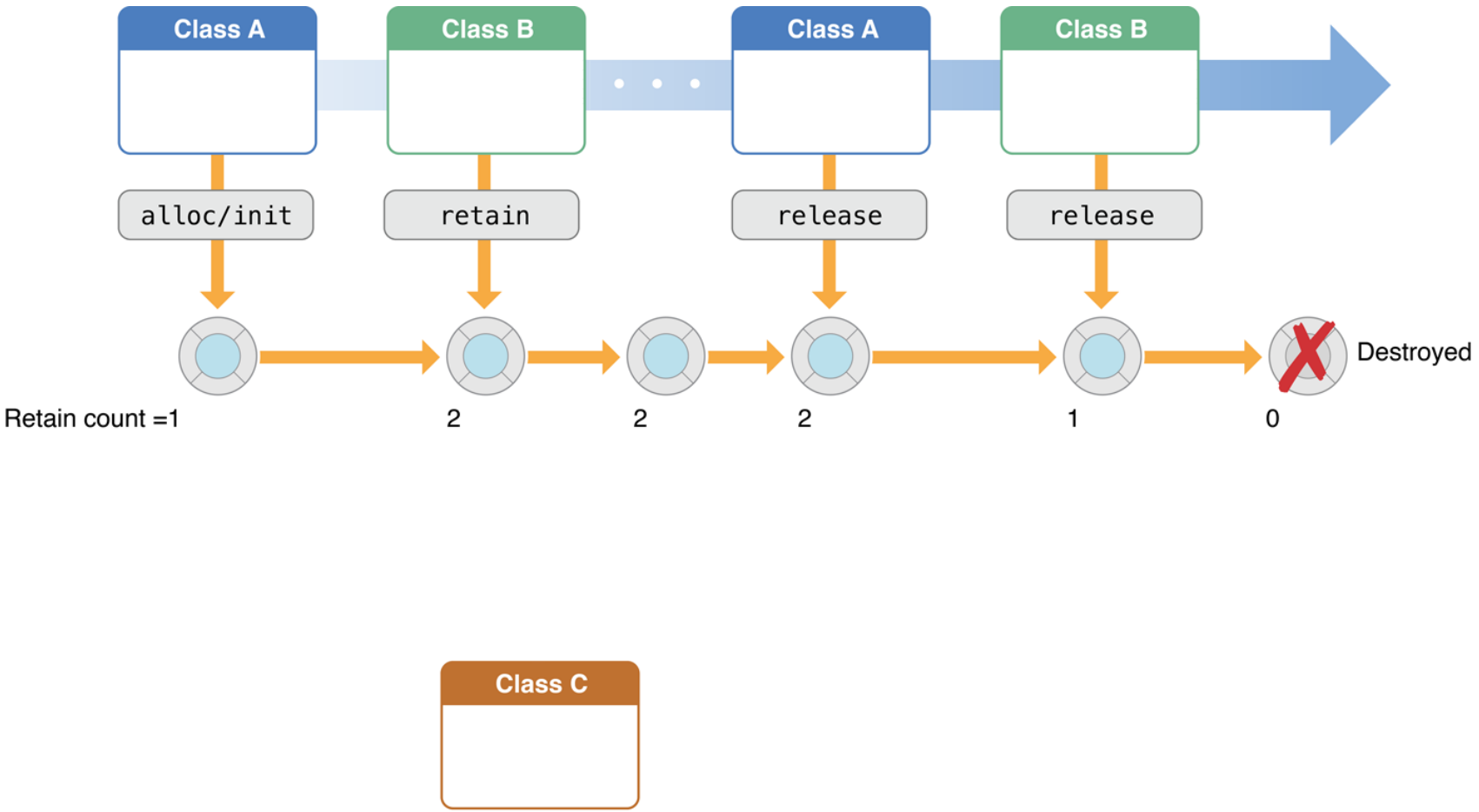


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

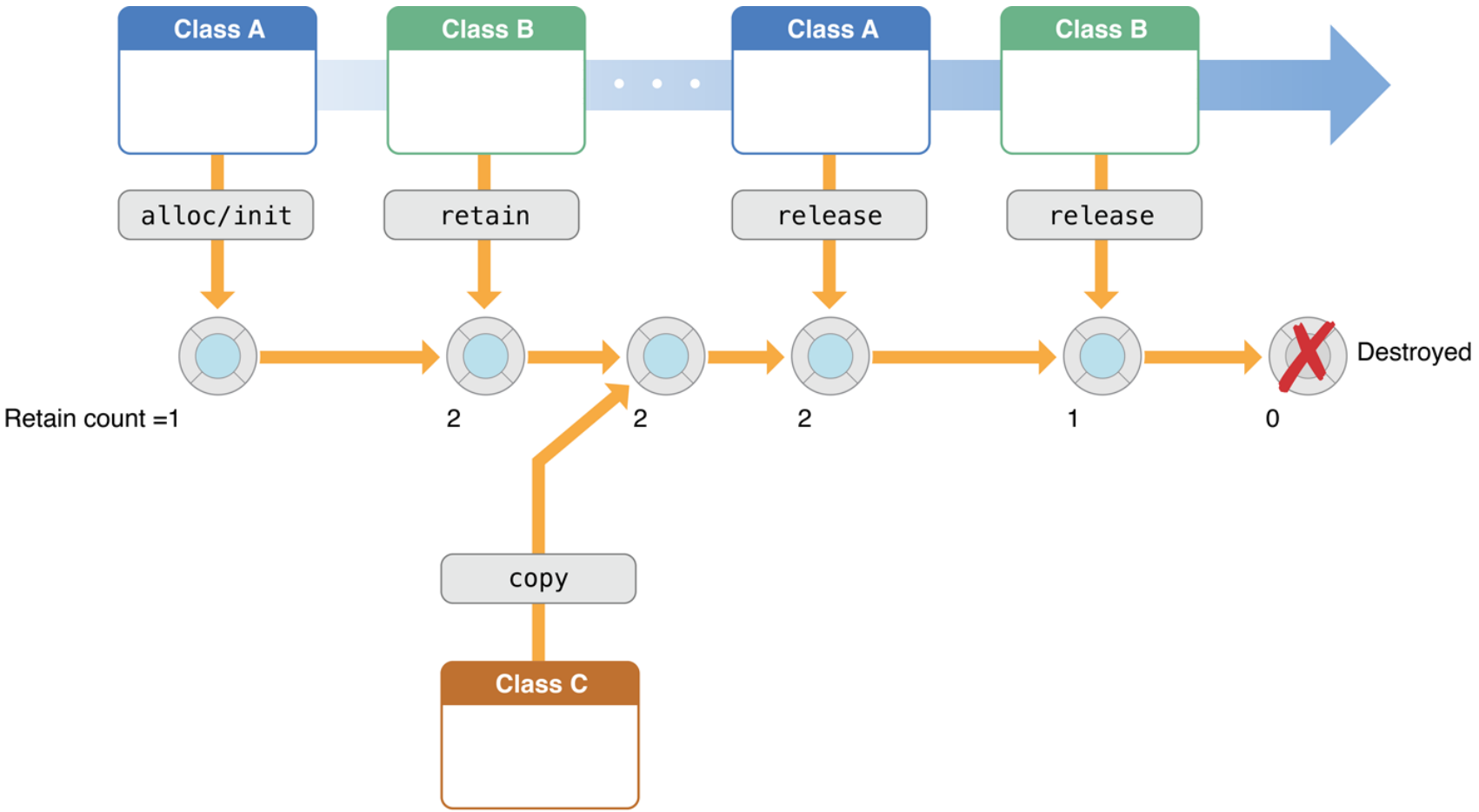


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

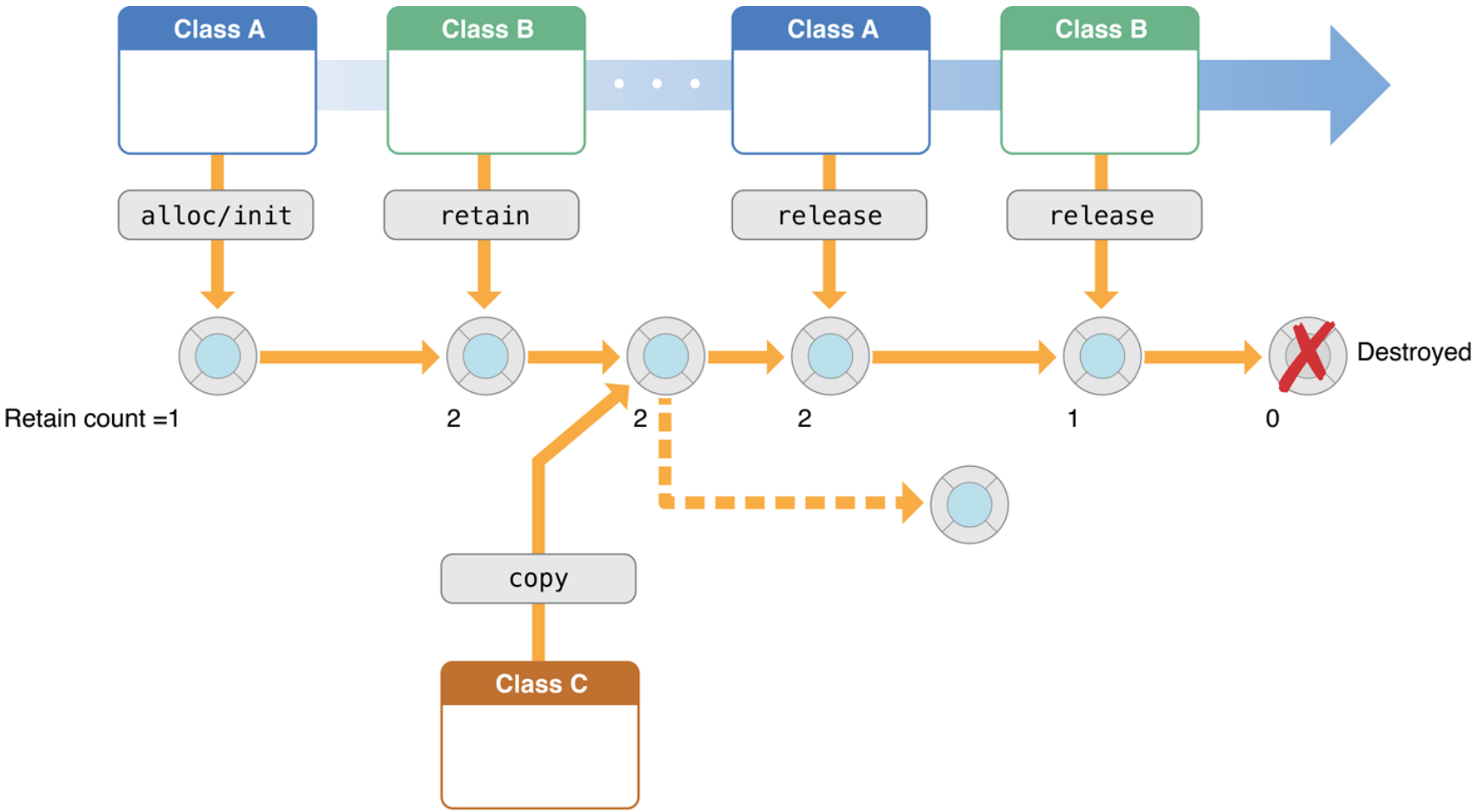


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

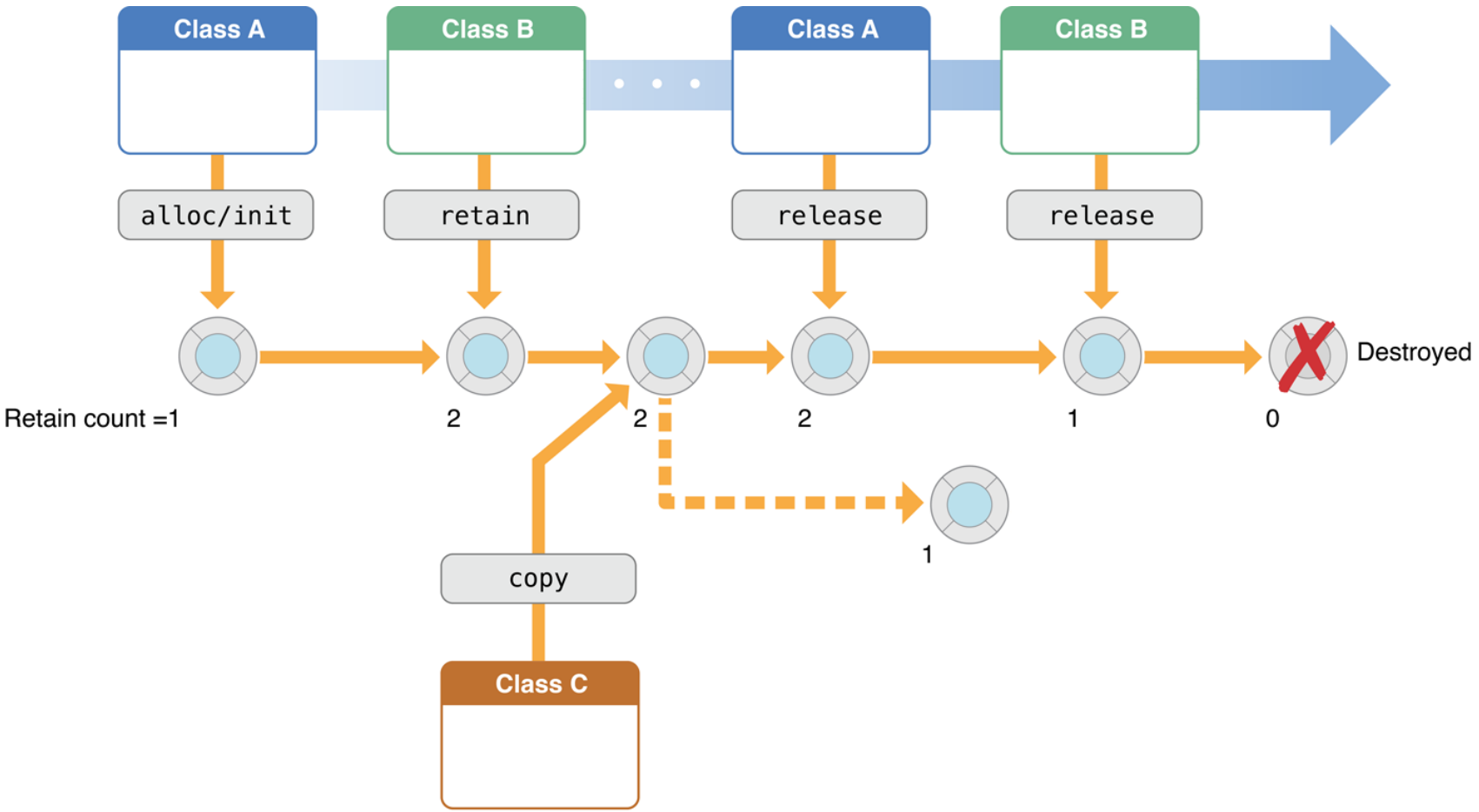


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

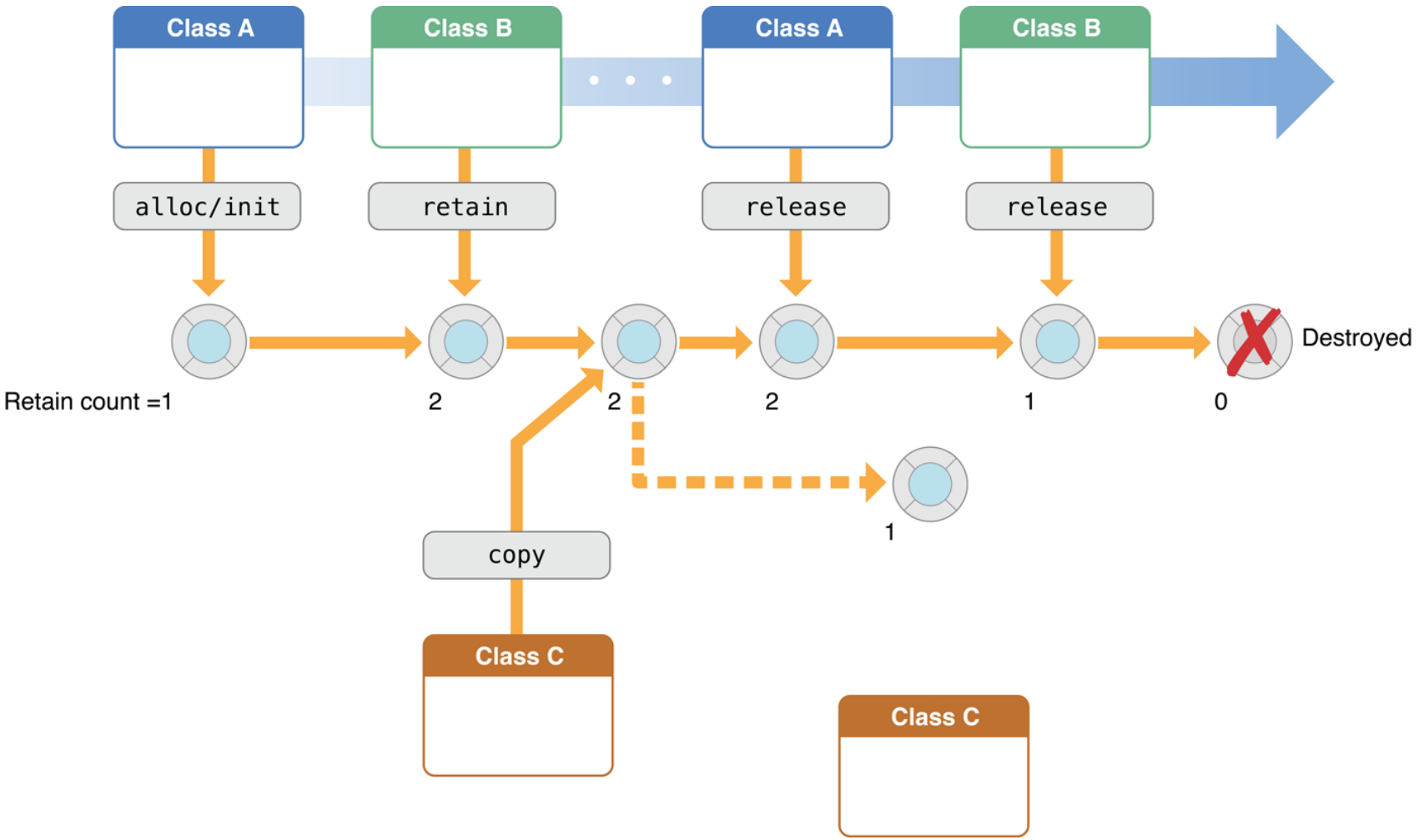


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

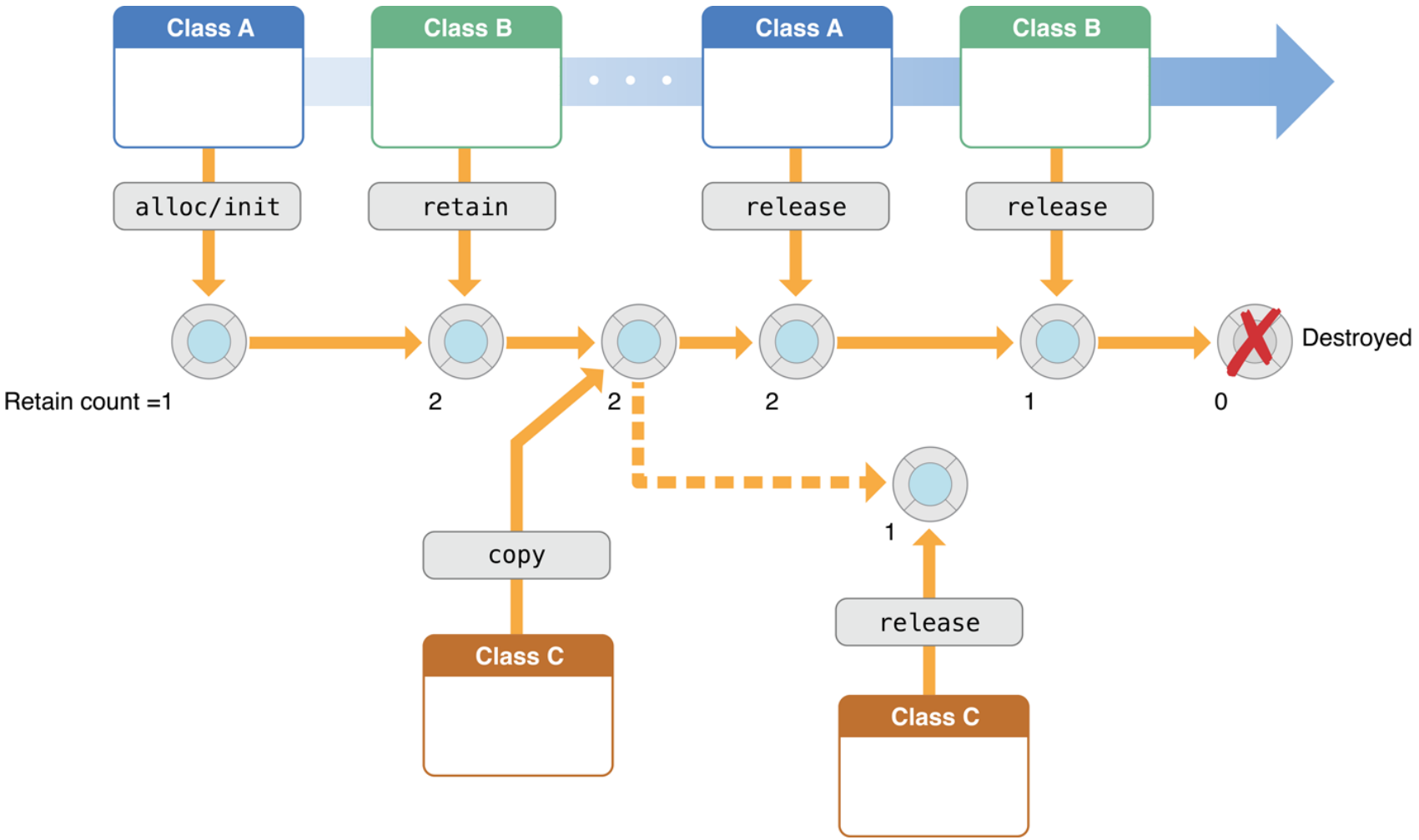


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

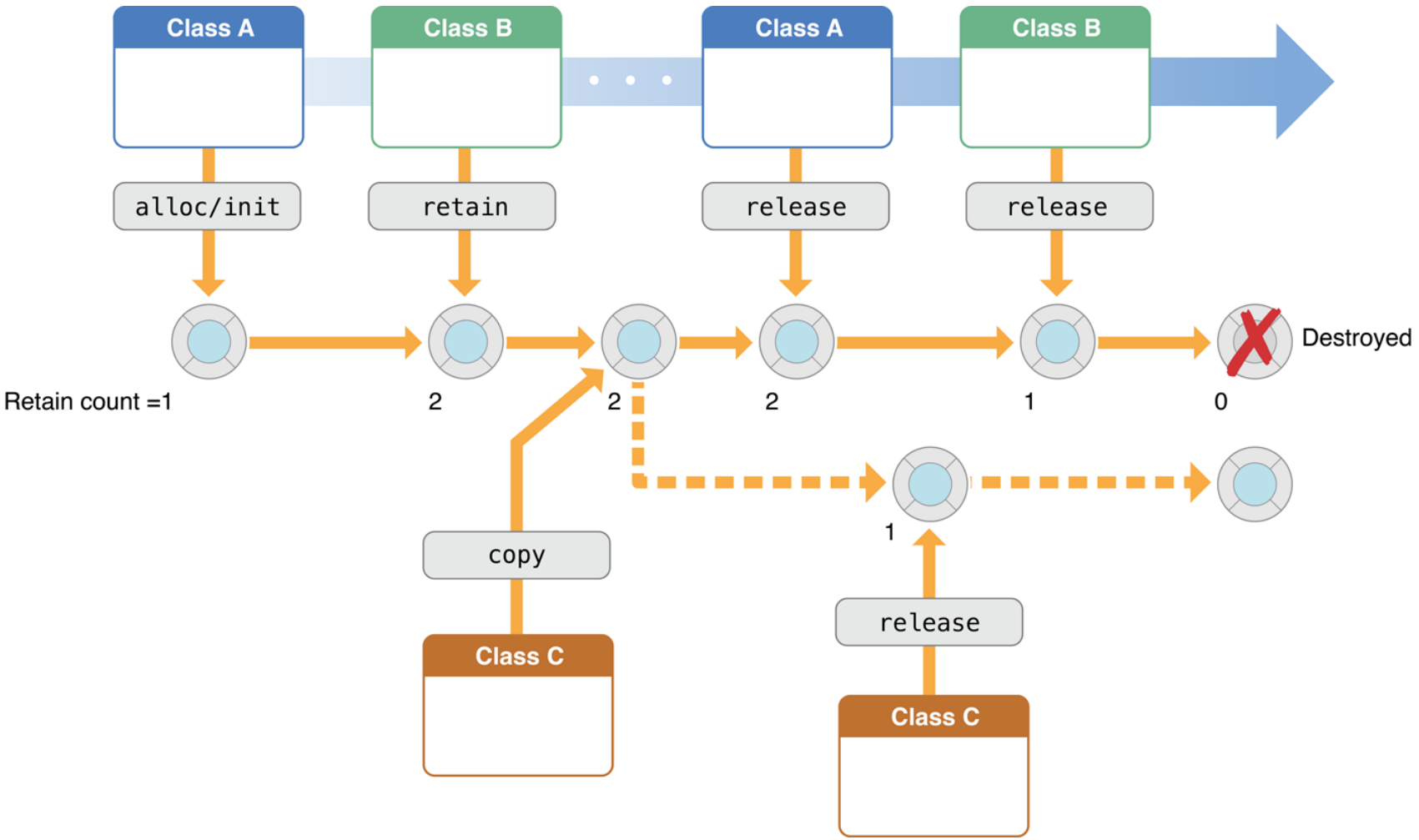


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

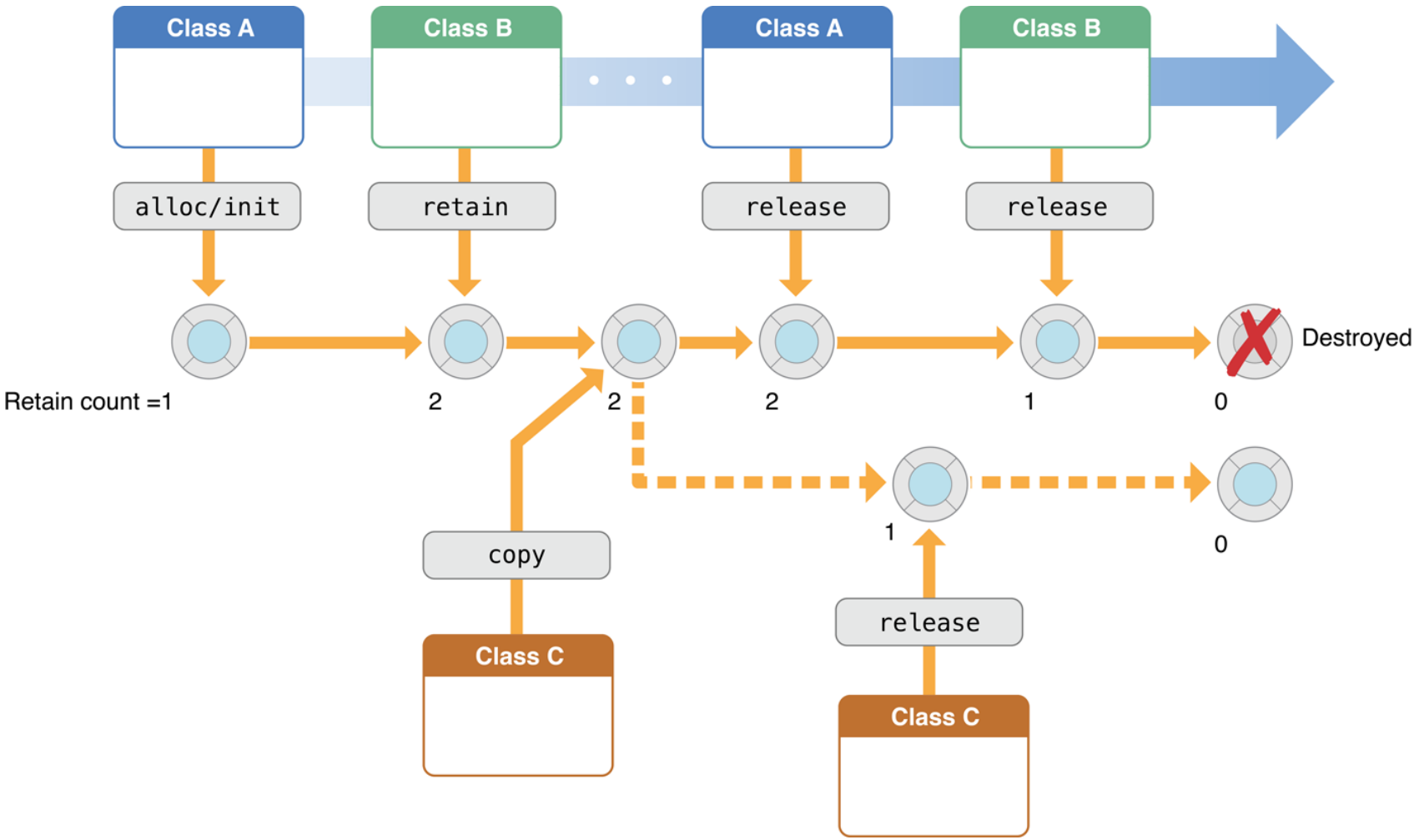


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

# Übersicht Reference Counting[3]

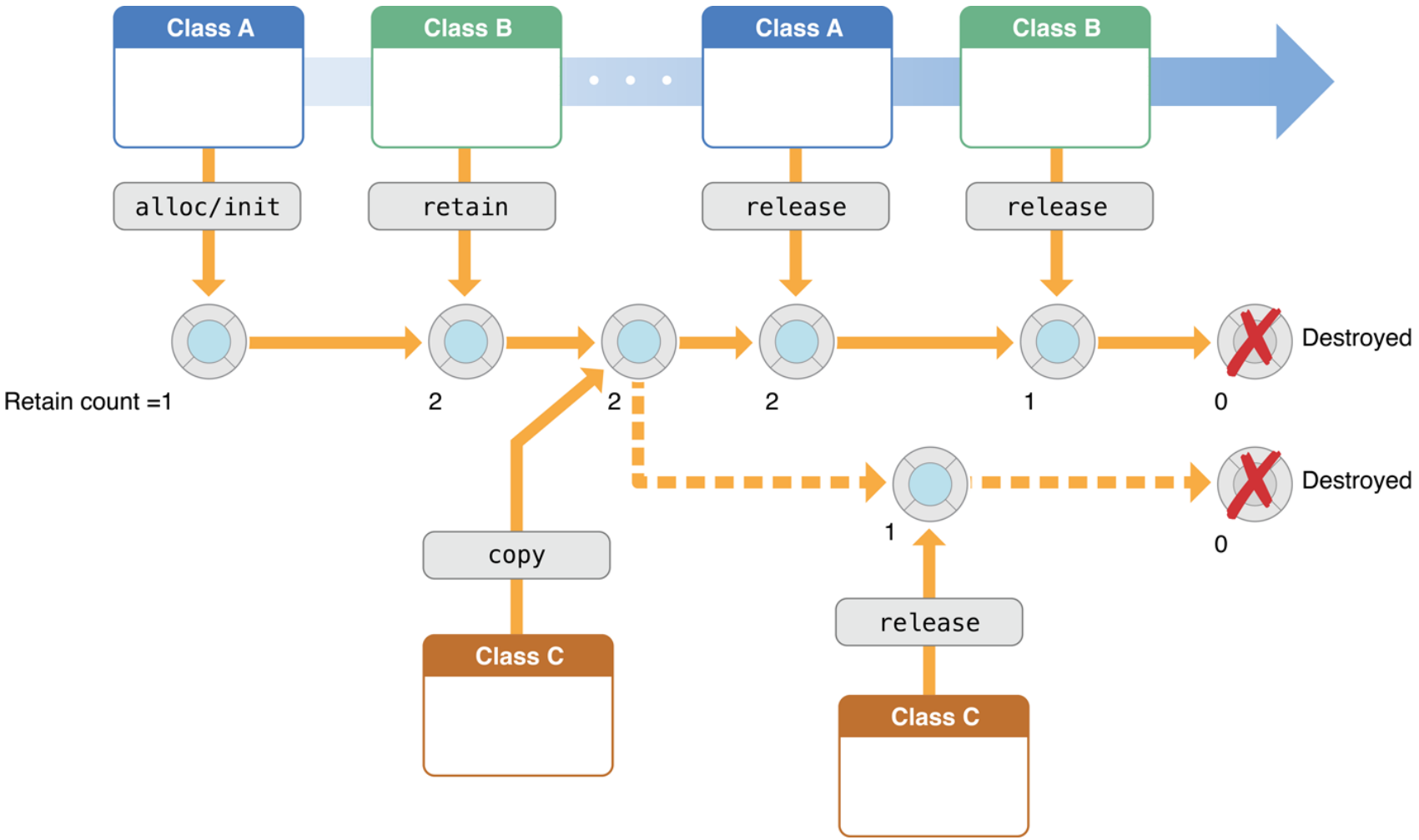


**Abbildung 1:** Speicherverwaltung mittels Reference Counting. [3]

## Grundlegende Speichermanagement Regeln[4]

Speichermanagement-Modelle basieren im Allgemeinen auf dem **Besitz** eines Objektes.

## Grundlegende Speichermanagement Regeln[4]

Speichermanagement-Modelle basieren im Allgemeinen auf dem **Besitz** eines Objektes.

1. **Jedes Objekt gehört dem Erzeuger.**
   *malloc(..)*, *calloc(..)*, *copy(..)*

## Grundlegende Speichermanagement Regeln[4]

Speichermanagement-Modelle basieren im Allgemeinen auf dem **Besitz** eines Objektes.

1. **Jedes Objekt gehört dem Erzeuger.**
   *malloc(..)*, *calloc(..)*, *copy(..)*
2. **Besitz kann geteilt/übernommen werden.**
   *retain(..)*

## Grundlegende Speichermanagement Regeln[4]

Speichermanagement-Modelle basieren im Allgemeinen auf dem **Besitz** eines Objektes.

1. **Jedes Objekt gehört dem Erzeuger.**
   *malloc(..)*, *calloc(..)*, *copy(..)*
2. **Besitz kann geteilt/übernommen werden.**
   *retain(..)*
3. **Besitz aufgeben, wenn Objekt nicht mehr benötigt wird.**
   *release(..)*

## Grundlegende Speichermanagement Regeln[4]

Speichermanagement-Modelle basieren im Allgemeinen auf dem **Besitz** eines Objektes.

1. **Jedes Objekt gehört dem Erzeuger.**
   *malloc(..)*, *calloc(..)*, *copy(..)*
2. **Besitz kann geteilt/übernommen werden.**
   *retain(..)*
3. **Besitz aufgeben, wenn Objekt nicht mehr benötigt wird.**
   *release(..)*
4. **Keinen Besitz aufgeben, den man nicht besitzt.**

## Typische Anwendungsbereiche[2]

Drei Szenarios:

## Typische Anwendungsbereiche[2]

Drei Szenarios:

- Objekt-**Lebenszeit-Management** für gemeinsam genutzte Objekte:
  *Speichermanagement.*

## Typische Anwendungsbereiche[2]

Drei Szenarios:

- Objekt-**Lebenszeit-Management** für gemeinsam genutzte Objekte:
  *Speichermanagement.*
- **Einschränkungs-Management** für Beziehungen von gemeinsam genutzten Objekten:
  *Limitierung der Anzahl an Referenzen → Biologie/Chemie.*

## Typische Anwendungsbereiche[2]

Drei Szenarios:

- Objekt-**Lebenszeit-Management** für gemeinsam genutzte Objekte:
  *Speichermanagement.*
- **Einschränkungs-Management** für Beziehungen von gemeinsam genutzten Objekten:
  *Limitierung der Anzahl an Referenzen → Biologie/Chemie.*
- **Metadaten**:
  *Zählen der Instanzen einer Klasse (Debugging/Profiling).*

# Problem: Zyklische Datenstrukturen[5]

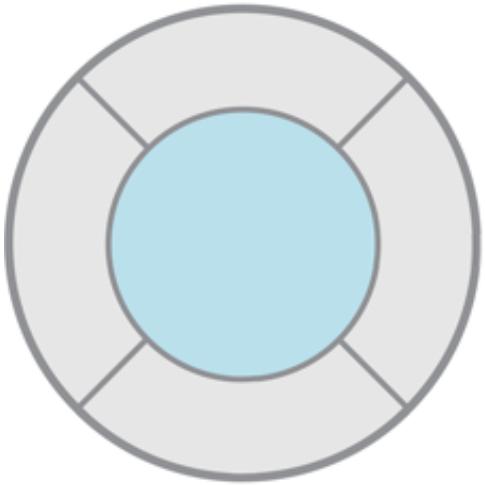

**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

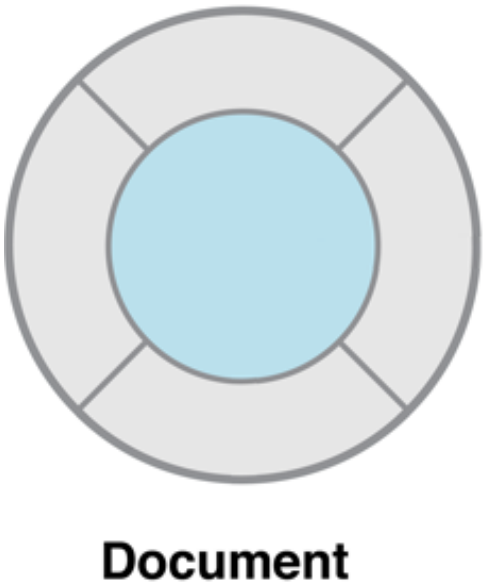


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

## Problem: Zyklische Datenstrukturen[5]

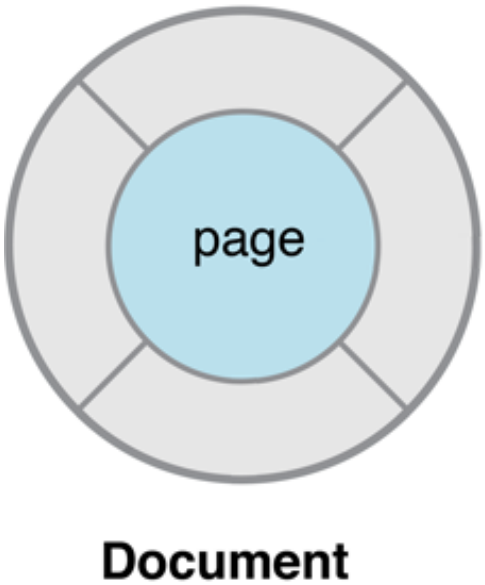


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

## Problem: Zyklische Datenstrukturen[5]

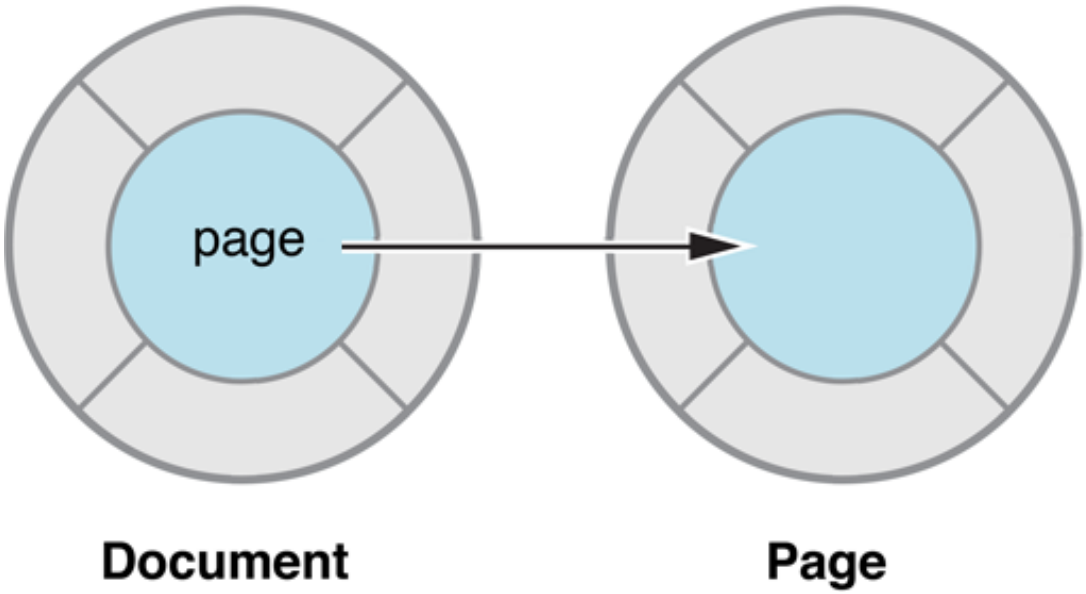


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

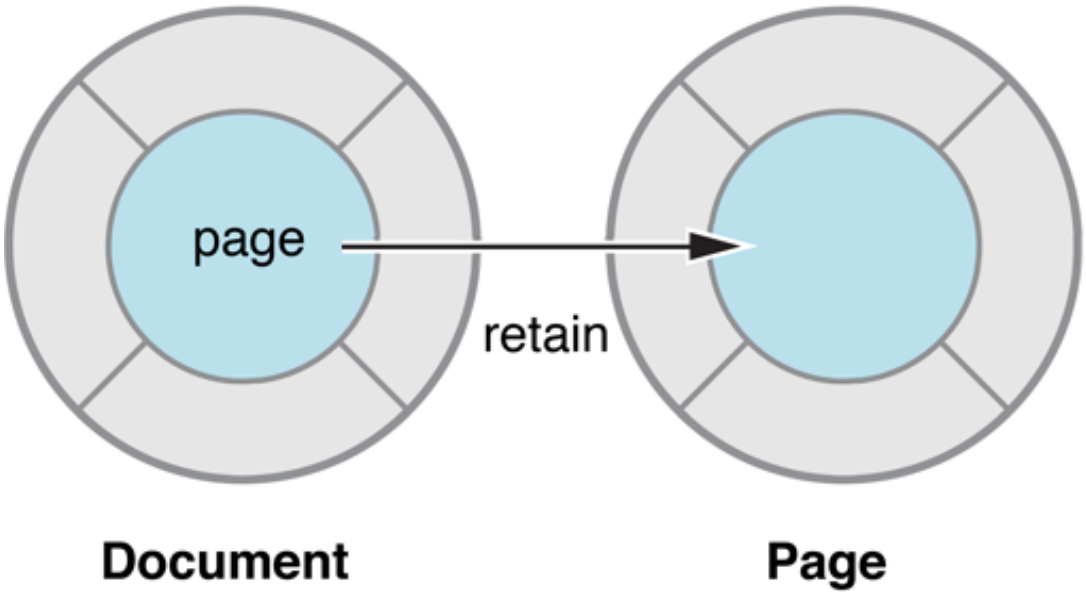


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

## Problem: Zyklische Datenstrukturen[5]

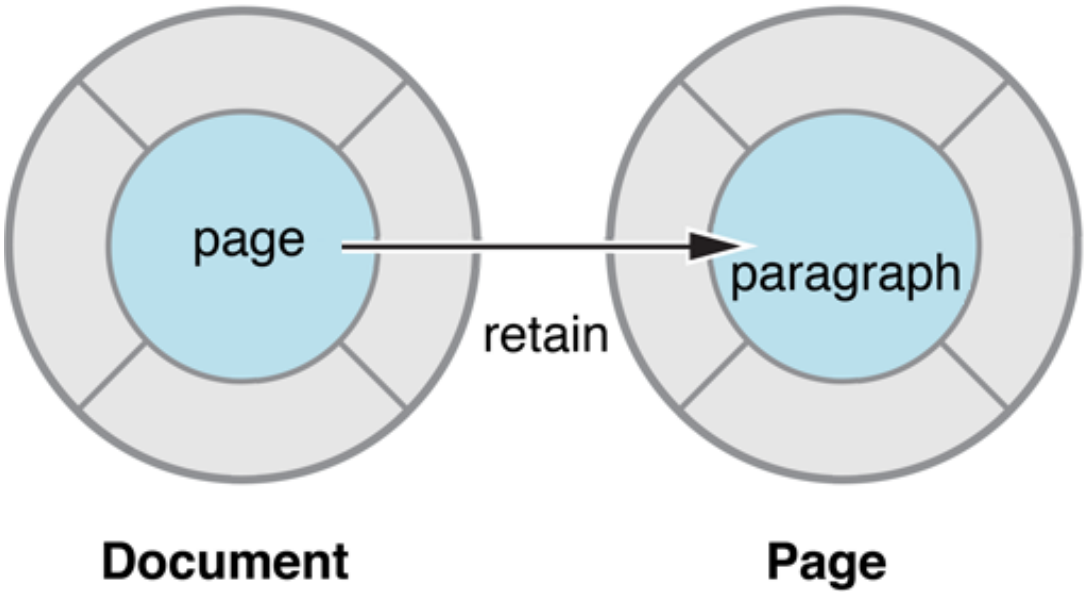


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

## Problem: Zyklische Datenstrukturen[5]

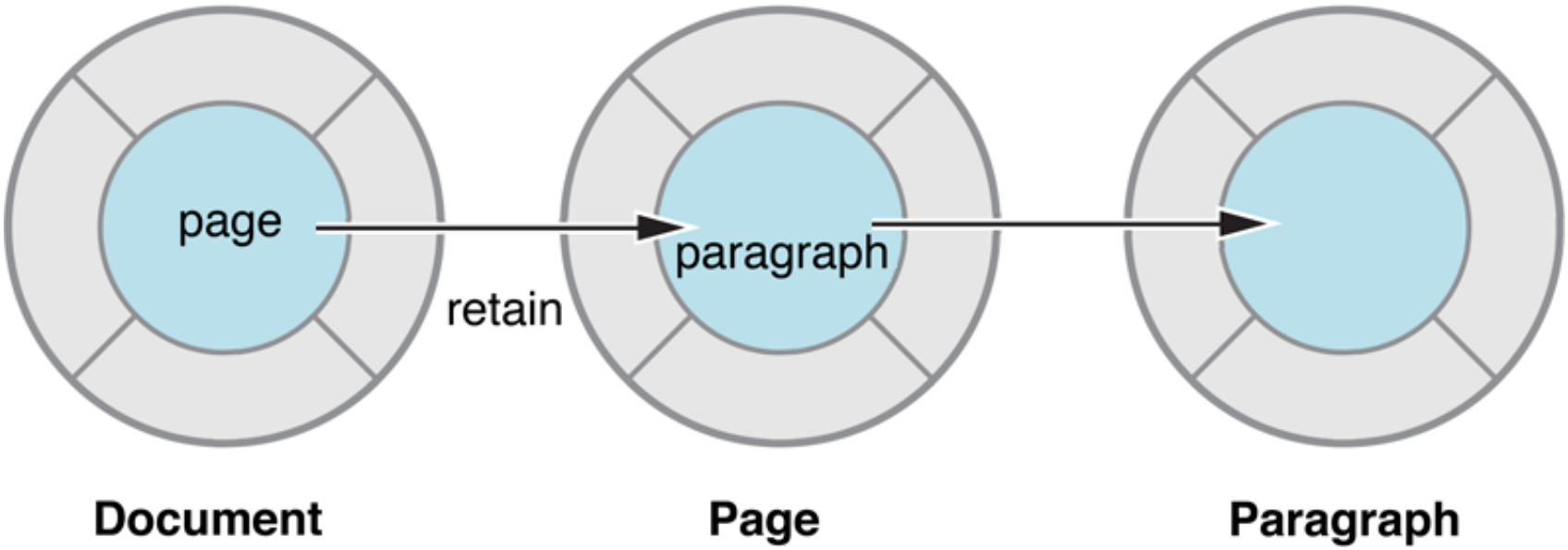


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

## Problem: Zyklische Datenstrukturen[5]

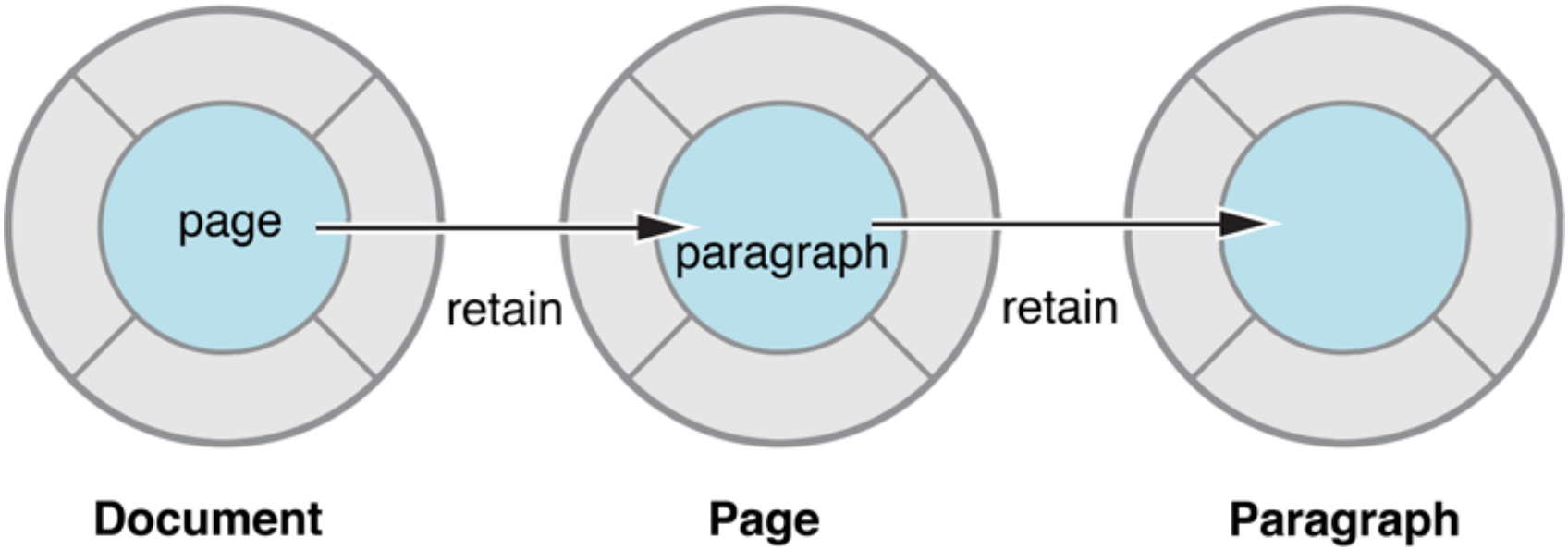


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

## Problem: Zyklische Datenstrukturen[5]

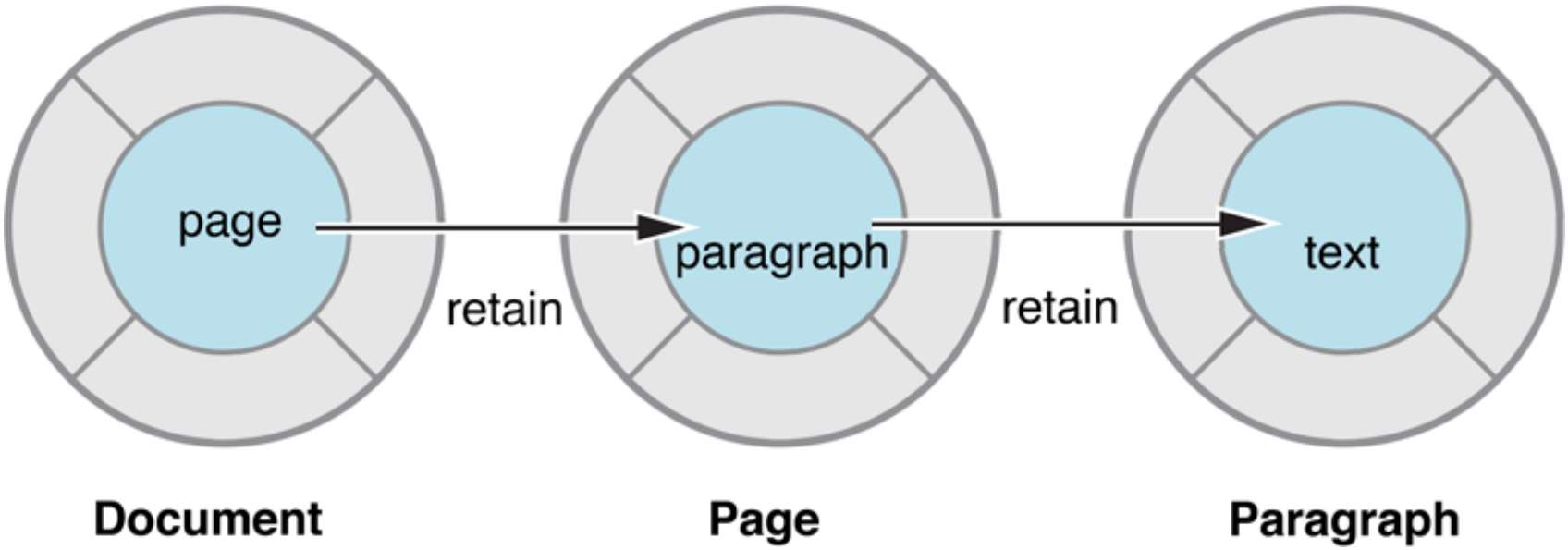


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

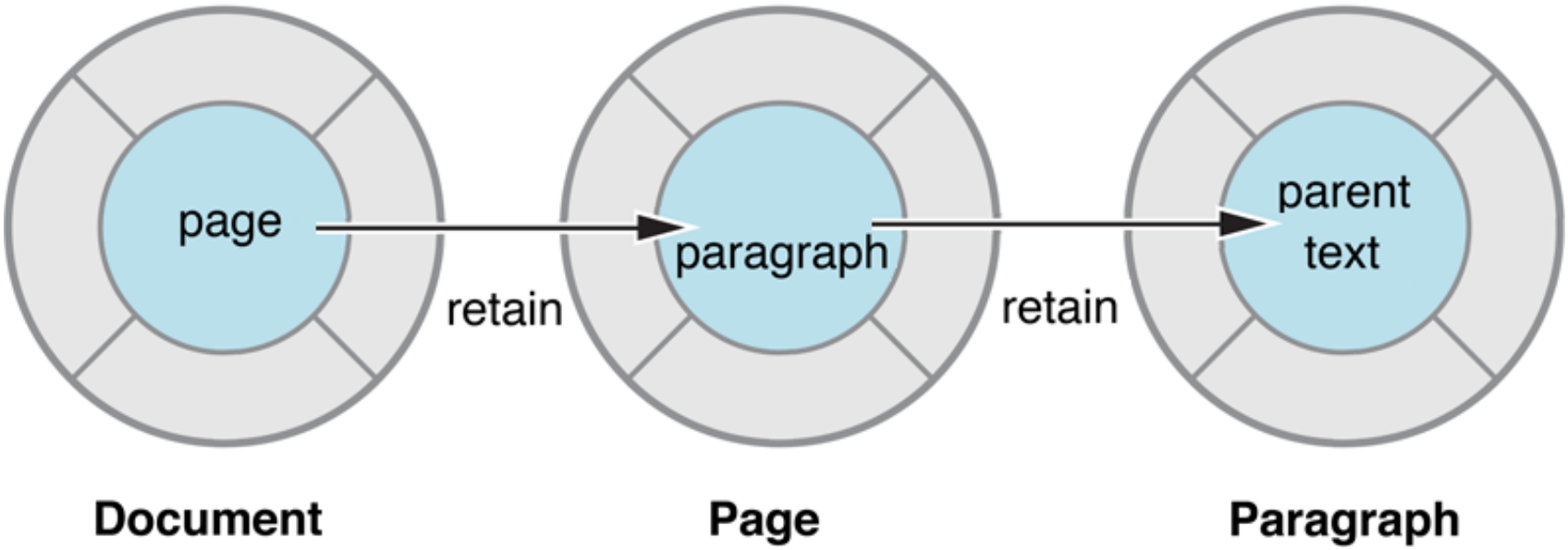


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

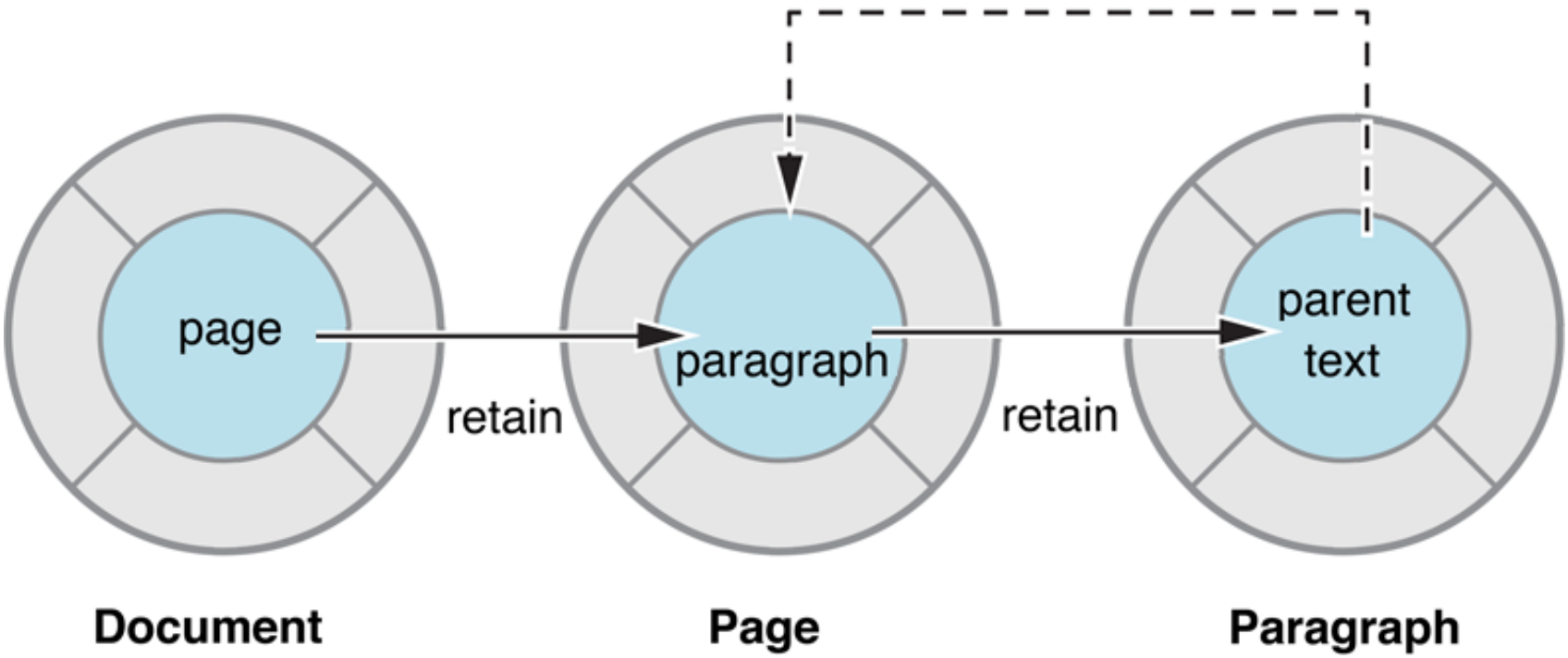


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

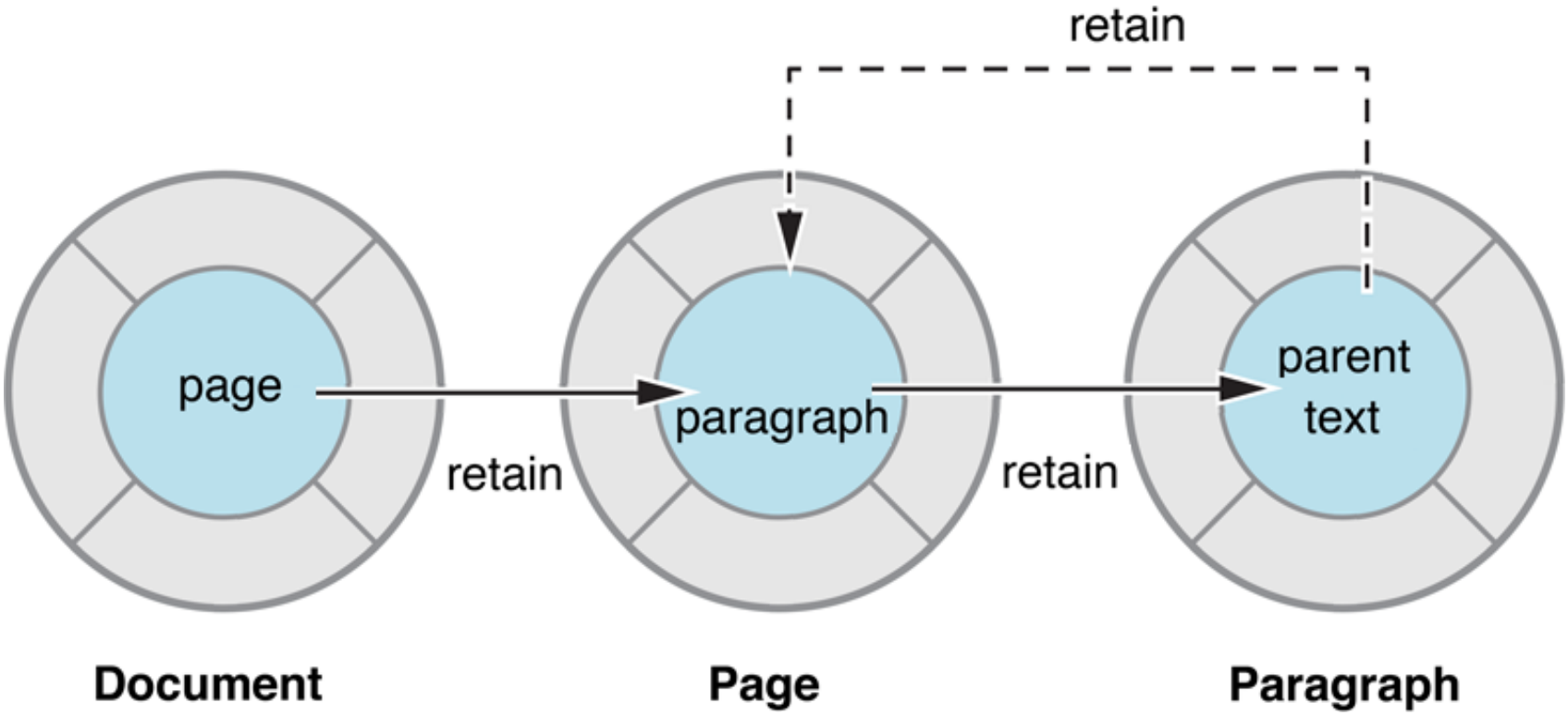


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

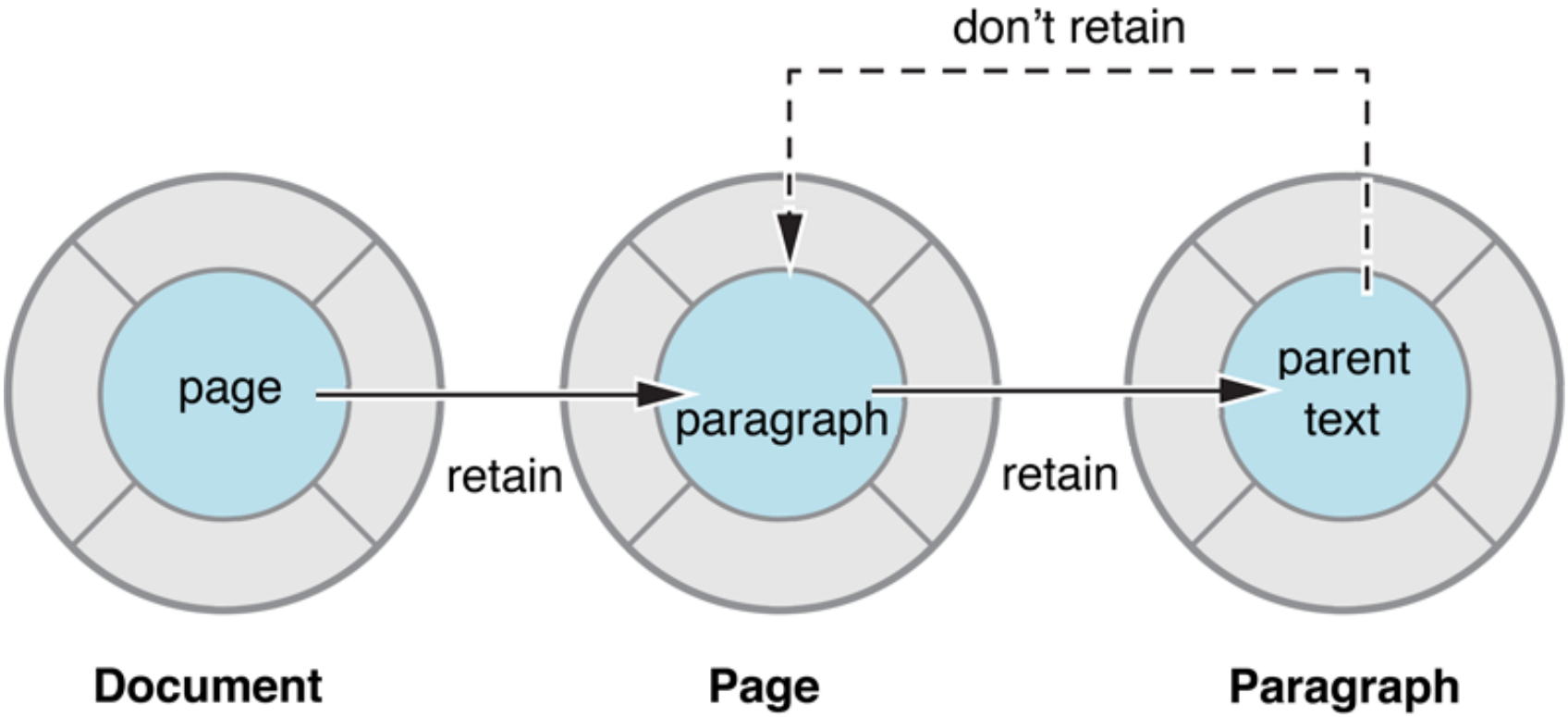


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

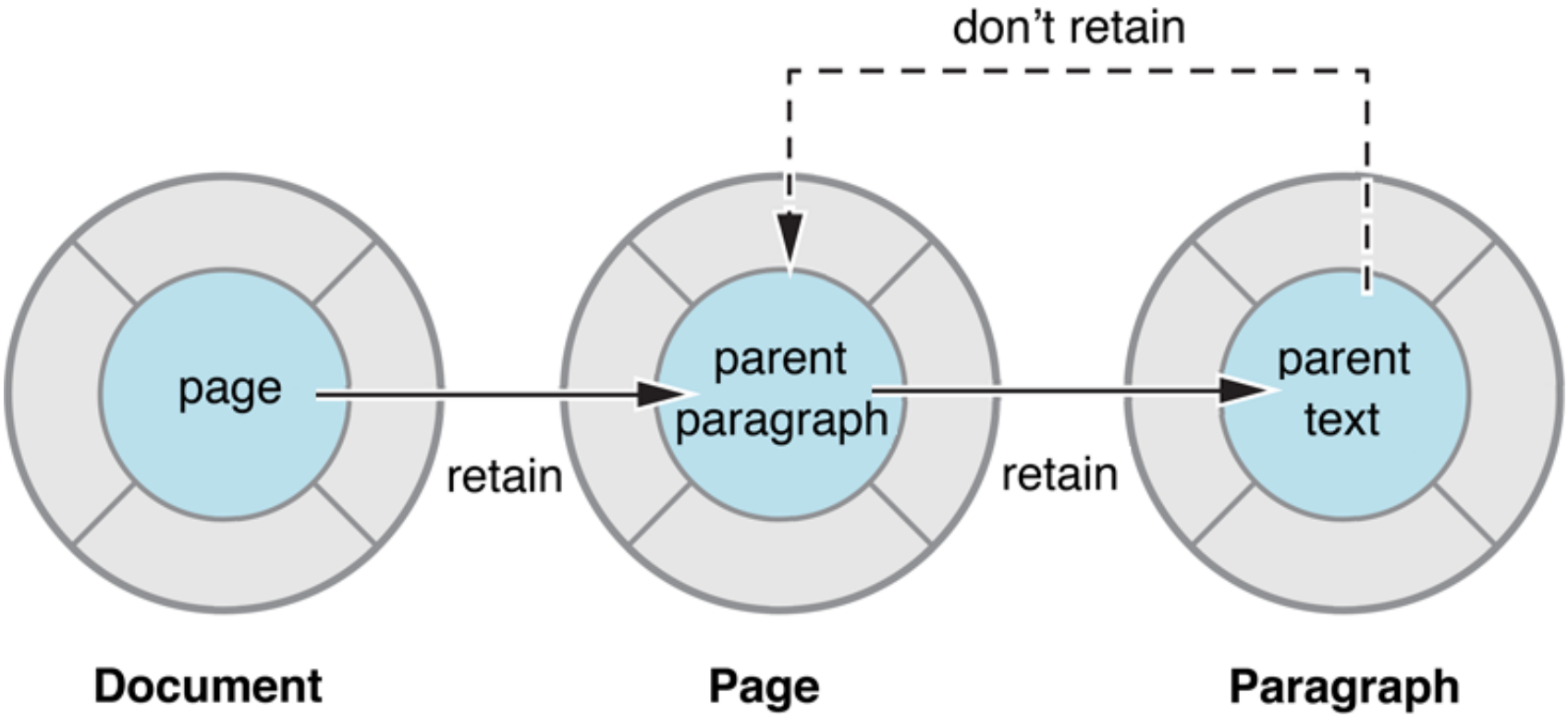


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

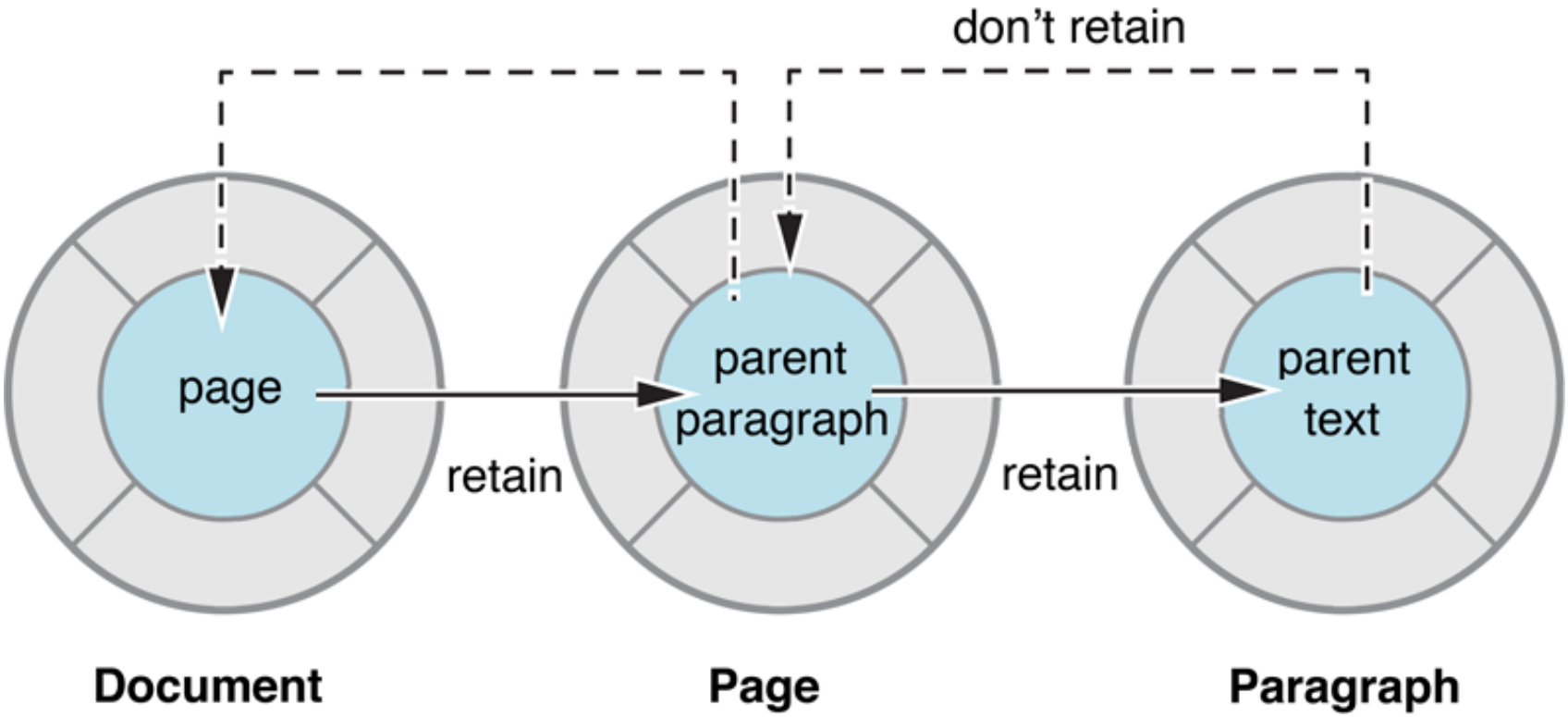


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

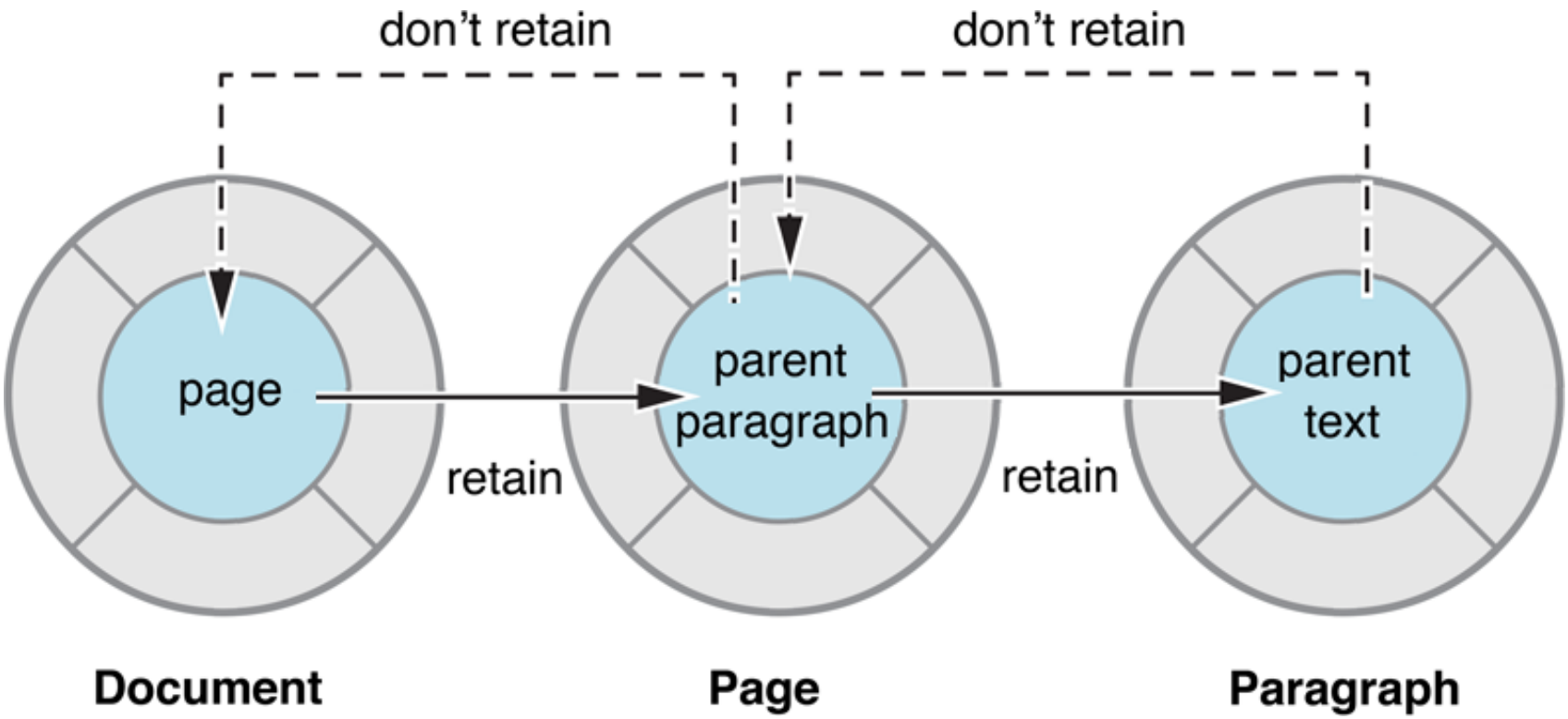


**Abbildung 2:** Zyklische Datenstrukturen beim Reference Counting. [5]

# Problem: Zyklische Datenstrukturen[5]

Lösung:

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.
  - **Vorsicht: Keine Garantie auf valide Referenz.**

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.
  - **Vorsicht: Keine Garantie auf valide Referenz.**
- Verwendung von ***strong*-Referenzen**.

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.
  - **Vorsicht: Keine Garantie auf valide Referenz.**
- Verwendung von ***strong*-Referenzen**.
  - Notwendig, damit Objekte nicht automatisch freigegeben werden.

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.
  - **Vorsicht: Keine Garantie auf valide Referenz.**
- Verwendung von ***strong*-Referenzen**.
  - Notwendig, damit Objekte nicht automatisch freigegeben werden.
- Neue Regel einführen:

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.
  - **Vorsicht: Keine Garantie auf valide Referenz.**
- Verwendung von ***strong*-Referenzen**.
  - Notwendig, damit Objekte nicht automatisch freigegeben werden.
- Neue Regel einführen:
  - *parent*-Objekte referenzieren ihre *children*-Objekte mit *strong*-Referenzen.

## Problem: Zyklische Datenstrukturen[5]

Lösung:

- Verwendung von ***weak*-Referenzen**.
  - Kein Besitz vom *parent*-Objekt fordern.
  - *retain*-Zähler bleibt unverändert.
  - **Vorsicht: Keine Garantie auf valide Referenz.**
- Verwendung von ***strong*-Referenzen**.
  - Notwendig, damit Objekte nicht automatisch freigegeben werden.
- Neue Regel einführen:
  - *parent*-Objekte referenzieren ihre *children*-Objekte mit *strong*-Referenzen.
  - *children*-Objekte referenzieren ihre *parent*-Objekte mit *weak*-Referenzen.

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

[3] http://docs.python.org/2/extending/extending.html
[4] https://wiki.gnome.org/Projects/Vala/ReferenceHandling

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**

[3] http://docs.python.org/2/extending/extending.html
[4] https://wiki.gnome.org/Projects/Vala/ReferenceHandling

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**
- **Cocoa**
  - Automatic Garbage Collection

[3] http://docs.python.org/2/extending/extending.html
[4] https://wiki.gnome.org/Projects/Vala/ReferenceHandling

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**
- **Cocoa**
  - Automatic Garbage Collection
- **GObject**
  - **atomic operations** für sicheres Multithreading

[3] http://docs.python.org/2/extending/extending.html
[4] https://wiki.gnome.org/Projects/Vala/ReferenceHandling

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**
- **Cocoa**
  - Automatic Garbage Collection
- **GObject**
  - **atomic operations** für sicheres Multithreading
- **Python**[3]

[3] `http://docs.python.org/2/extending/extending.html`
[4] `https://wiki.gnome.org/Projects/Vala/ReferenceHandling`

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**
- **Cocoa**
  - Automatic Garbage Collection
- **GObject**
  - **atomic operations** für sicheres Multithreading
- **Python**[3]
- **VALA**[4]

[3] `http://docs.python.org/2/extending/extending.html`
[4] `https://wiki.gnome.org/Projects/Vala/ReferenceHandling`

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**
- **Cocoa**
  - Automatic Garbage Collection
- **GObject**
  - **atomic operations** für sicheres Multithreading
- **Python**[3]
- **VALA**[4]
- File systems

[3] `http://docs.python.org/2/extending/extending.html`
[4] `https://wiki.gnome.org/Projects/Vala/ReferenceHandling`

## Sprachunterstützung

Beispiele, die das *Reference Counting*-Konzept unterstützen:

- **c++11**
- **Cocoa**
  - Automatic Garbage Collection
- **GObject**
  - **atomic operations** für sicheres Multithreading
- **Python**[3]
- **VALA**[4]
- File systems
- COM, Delphi, Perl, Tcl, u.v.m.

[3] `http://docs.python.org/2/extending/extending.html`
[4] `https://wiki.gnome.org/Projects/Vala/ReferenceHandling`

## Praktisches Beispiel: GLib Singelton Design Pattern[7]

```
static MySingleton * the_singleton = NULL;
static GObject * my_singleton_constructor(GType type,
            guint n_construct_params,
            GObjectConstructParam * construct_params)
{
    GObject * object = NULL;
    if (!the_singleton)
    {
        object = G_OBJECT_CLASS(parent_class)->constructor(type,
                    n_construct_params, construct_params);
        the_singleton = MY_SINGLETON(object);
    }
    else
        object = g_object_ref(G_OBJECT(the_singleton));

    return object;
}
```

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.
  - Zwei *auto_ptr*-Objekten darf nicht das gleiche Objekt gehören.

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.
  - Zwei *auto_ptr*-Objekten darf nicht das gleiche Objekt gehören.
  - Bei Zuweisung **wechselt** der Besitz des Objektes.

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.
  - Zwei *auto_ptr*-Objekten darf nicht das gleiche Objekt gehören.
  - Bei Zuweisung **wechselt** der Besitz des Objektes.
  - Deprecated → *std::unique_ptr*.

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.
  - Zwei *auto_ptr*-Objekten darf nicht das gleiche Objekt gehören.
  - Bei Zuweisung **wechselt** der Besitz des Objektes.
  - Deprecated → *std::unique_ptr*.

- std::unique_ptr *(c++11)*[6]

```
template <class T, class D = default_delete<T>>
    class unique_ptr;
```

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.
  - Zwei *auto_ptr*-Objekten darf nicht das gleiche Objekt gehören.
  - Bei Zuweisung **wechselt** der Besitz des Objektes.
  - Deprecated → *std::unique_ptr*.

- std::unique_ptr *(c++11)*[6]

```
template <class T, class D = default_delete<T>>
    class unique_ptr;
```

  - *stored-Pointer*: Zeiger auf den Speicherbereich.

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

- std::auto_ptr[5]

```
template <class X> class auto_ptr;
```

  - **Alleiniger** Besitz eines Objektes.
  - Zwei *auto_ptr*-Objekten darf nicht das gleiche Objekt gehören.
  - Bei Zuweisung **wechselt** der Besitz des Objektes.
  - Deprecated → *std::unique_ptr*.

- std::unique_ptr *(c++11)*[6]

```
template <class T, class D = default_delete<T>>
    class unique_ptr;
```

  - *stored-Pointer*: Zeiger auf den Speicherbereich.
  - *stored-Deleter*: Mit dem *stored-Pointer* als Argument aufrufbares *Deleter*-Objekt.

[5] http://www.cplusplus.com/reference/memory/auto_ptr/
[6] http://www.cplusplus.com/reference/memory/unique_ptr/

## Erweiterte *Reference Counting*-Konzepte

Lösung für das Motivationsbeispiel:

```
#include <memory>

int main(int argc, char * argv[])
{
    std::auto_ptr<int> p1;
    std::auto_ptr<int> p2(new int);
    std::auto_ptr<int> p3(new int);

    p3 = p2;    // Original p3 has been lost.
                // p3 owns the object pointed to by p2.
                // data will be automatically deleted when
                // auto_ptrs go out of visibility scope.

    return 0;
}
```

## Erweiterte *Reference Counting*-Konzepte

- **Automatic Reference Counting (ARC)[6]**

## Erweiterte *Reference Counting*-Konzepte

- **Automatic Reference Counting (ARC)[6]**
  - **Objective C**

## Erweiterte *Reference Counting*-Konzepte

- **Automatic Reference Counting (ARC)[6]**
  - **Objective C**
  - Compiler **Feature**

## Erweiterte *Reference Counting*-Konzepte

- **Automatic Reference Counting (ARC)[6]**
  - **Objective C**
  - Compiler **Feature**
  - Fokussierung auf *interessanten* Code

## Erweiterte *Reference Counting*-Konzepte

- **Automatic Reference Counting (ARC)[6]**
  - **Objective C**
  - Compiler **Feature**
  - Fokussierung auf *interessanten* Code

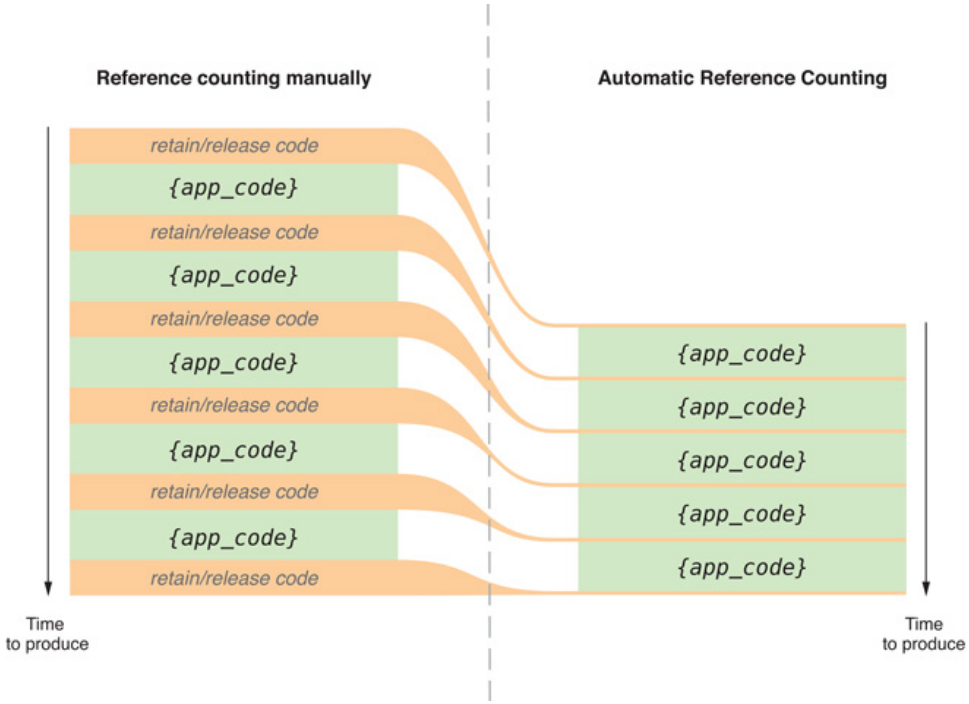


**Abbildung 3:** Zeitreduzierung durch ARC.[6]

## Performancevergleich für drei Speichermanagementansätze

Folgende Ansätze werden untersucht:

## Performancevergleich für drei Speichermanagementansätze

Folgende Ansätze werden untersucht:

1. **Standard C**: Eigene Speicherverwaltung
   *malloc(..), calloc(..), free(..)*

## Performancevergleich für drei Speichermanagementansätze

Folgende Ansätze werden untersucht:

1. **Standard C**: Eigene Speicherverwaltung
   *malloc(..), calloc(..), free(..)*
2. **Reference Counting** nach Jean-David Gadina:
   *alloc(..), retain(..), release(..)*

## Performancevergleich für drei Speichermanagementansätze

Folgende Ansätze werden untersucht:

1. **Standard C**: Eigene Speicherverwaltung
   *malloc(..), calloc(..), free(..)*
2. **Reference Counting** nach Jean-David Gadina:
   *alloc(..), retain(..), release(..)*
3. **Reference Counting** mit **GLib**:
   *my_object_new(..), g_object_ref(..), g_object_unref(..)*

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

[7] http://wr.informatik.uni-hamburg.de/

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe

[7] `http://wr.informatik.uni-hamburg.de/`

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe
- **1.000.000 - 10.000.000** Iterationsschleifen

[7] `http://wr.informatik.uni-hamburg.de/`

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe
- **1.000.000 - 10.000.000** Iterationsschleifen
- **10** Wiederholungen

[7] `http://wr.informatik.uni-hamburg.de/`

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe
- **1.000.000 - 10.000.000** Iterationsschleifen
- **10** Wiederholungen
  → Ergebnis: Mittelwert + Standardabweichung

[7] `http://wr.informatik.uni-hamburg.de/`

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe
- **1.000.000 - 10.000.000** Iterationsschleifen
- **10** Wiederholungen
  $\rightarrow$ Ergebnis: Mittelwert + Standardabweichung
- Keine Optimierungen

[7] `http://wr.informatik.uni-hamburg.de/`

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe
- **1.000.000 - 10.000.000** Iterationsschleifen
- **10** Wiederholungen
  $\rightarrow$ Ergebnis: Mittelwert + Standardabweichung
- Keine Optimierungen

System/Plattform:

[7] `http://wr.informatik.uni-hamburg.de/`

## Performancevergleich für drei Speichermanagementansätze

Der Performancevergleich:

- **3** Funktionalitäten für eine int-Variable
  1. Speicherallokation
  2. Referenzieren
  3. Speicherfreigabe
- **1.000.000 - 10.000.000** Iterationsschleifen
- **10** Wiederholungen
  $\rightarrow$ Ergebnis: Mittelwert + Standardabweichung
- Keine Optimierungen

System/Plattform:

- WR-Cluster[7]

[7] `http://wr.informatik.uni-hamburg.de/`

# System/Plattform

## System/Plattform

- WR-Cluster mit 10 Knoten, jeder Knoten besitzt folgende Ausstattung
  - 2 Prozessoren (Intel Xeon Westmere 5650 @ 2.67GHz) mit jeweils 6 cores
  - 12 GByte DDR3 / PC1333 Hauptspeicher (System taktet mit 1333 MHz)

## System/Plattform

- WR-Cluster mit 10 Knoten, jeder Knoten besitzt folgende Ausstattung
  - 2 Prozessoren (Intel Xeon Westmere 5650 @ 2.67GHz) mit jeweils 6 cores
  - 12 GByte DDR3 / PC1333 Hauptspeicher (System taktet mit 1333 MHz)
- **1** Task pro Node

## System/Plattform

- WR-Cluster mit 10 Knoten, jeder Knoten besitzt folgende Ausstattung
  - 2 Prozessoren (Intel Xeon Westmere 5650 @ 2.67GHz) mit jeweils 6 cores
  - 12 GByte DDR3 / PC1333 Hauptspeicher (System taktet mit 1333 MHz)
- **1** Task pro Node
- **10** Batches

## Standard C: Eigenes Speichermanagement

Pseudo Code für *testWithoutReferenceCounting*-Funktion:

```
unsigned long i = 0;
int * referencePointer = NULL;
int ** ptrs = (int **)calloc(numIterations, sizeof(int *));

for(i = 0; i < numIterations; ++i)
    ptrs[i] = (int *)calloc(1, sizeof(int));    // 1. Allocation

for(i = 0; i < numIterations; ++i)
    referencePointer = ptrs[i];                 // 2. Retain

for(i = 0; i < numIterations; ++i)
    free(ptrs[i]);                              // 3. Release

free(ptrs);
```

# Standard C: Eigenes Speichermanagement

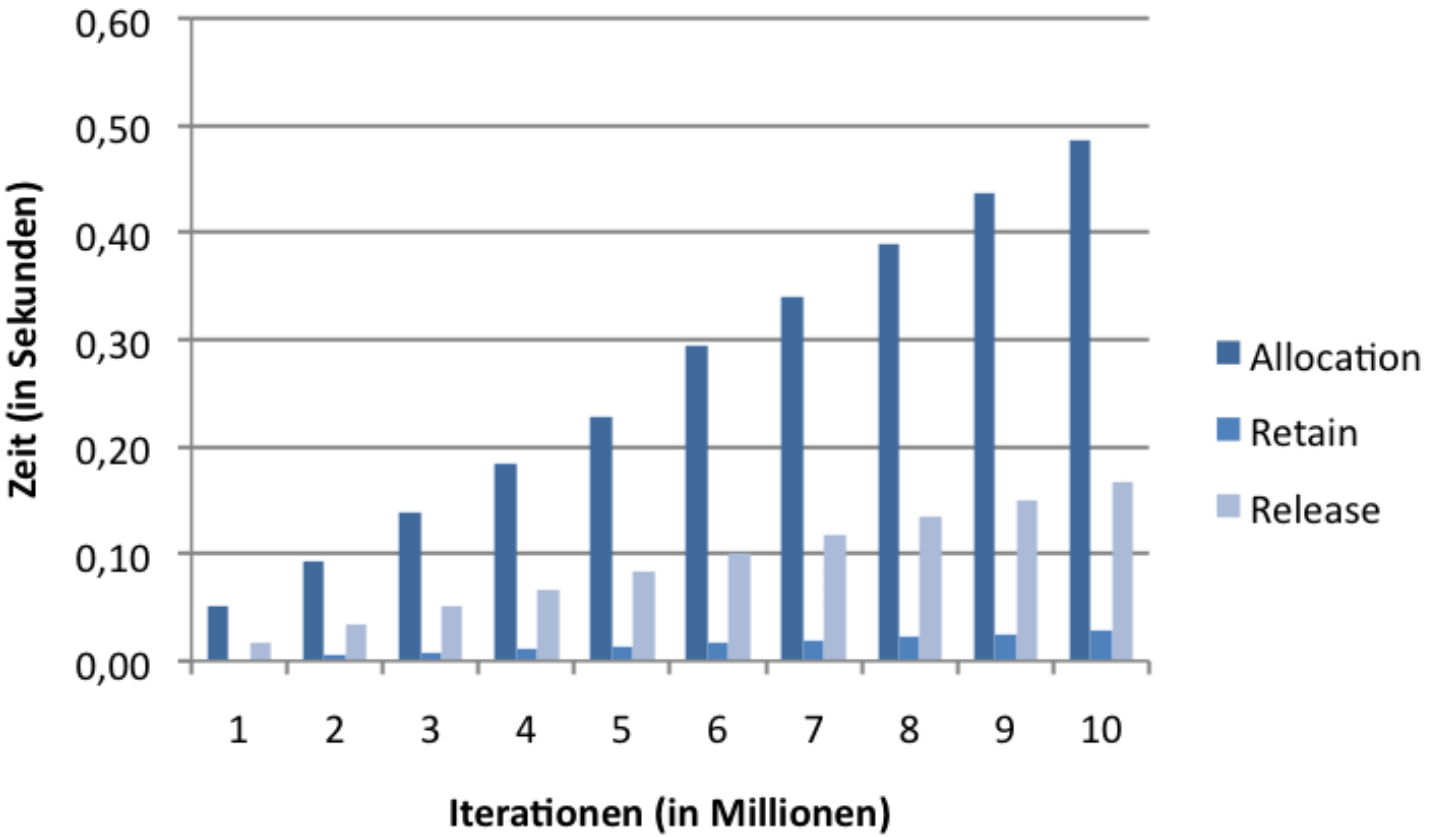


**Abbildung 4:** Performance: Standard C ohne Reference Counting.

# Standard C: Eigenes Speichermanagement

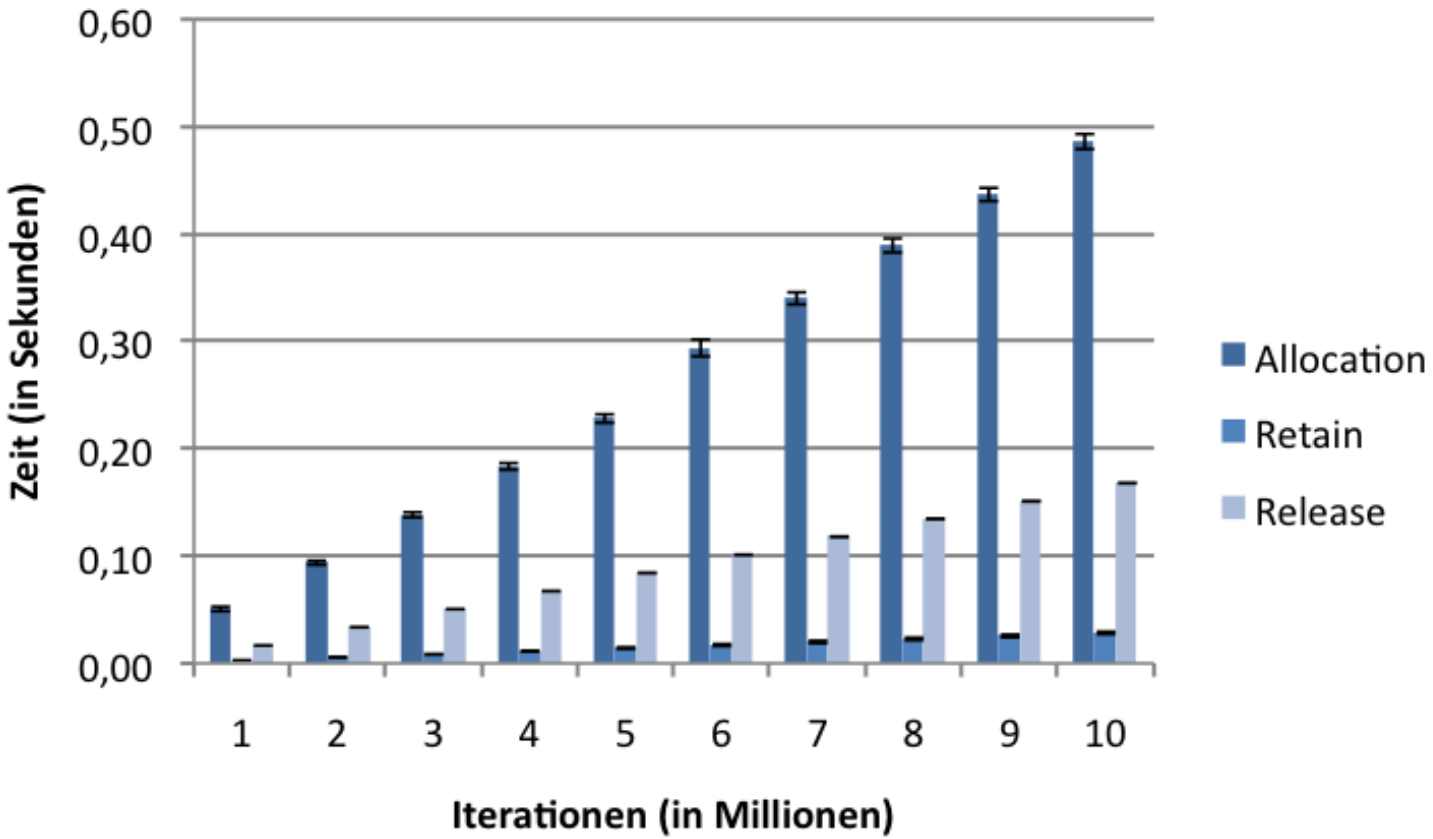


**Abbildung 4:** Performance: Standard C ohne Reference Counting.

## Reference Counting nach Jean-David Gadina[1]

Definition eines Speicherobjekts:

```
typedef struct
{
    unsigned int retainCount;
    void * data;
} MemoryObject;
```

## Reference Counting nach Jean-David Gadina[1]

Definition eines Speicherobjekts:

```
typedef struct
{
    unsigned int retainCount;
    void * data;
} MemoryObject;
```

Um welches Entwurfs-/Strukturmuster handelt es sich?

## Reference Counting nach Jean-David Gadina[1]

Definition eines Speicherobjekts:

```
typedef struct
{
    unsigned int retainCount;
    void * data;
} MemoryObject;
```

Um welches Entwurfs-/Strukturmuster handelt es sich?

→ Proxy (Stellvertreter)

## Reference Counting nach Jean-David Gadina[1]

Definition der *alloc*-Funktion:

```
void * alloc(size_t size)
{
    MemoryObject * o = NULL;
    char * ptr = NULL;

    o = (MemoryObject *)calloc(sizeof(MemoryObject) + size, 1);
    if(o != NULL)
    {
        ptr = (char *)o;
        ptr += sizeof(MemoryObject);
        o->retainCount = 1;
        o->data = ptr;
    }
    return (void *)ptr;
}
```

## Reference Counting nach Jean-David Gadina[1]

Definition der *retain*-Funktion:

```
void retain(void * ptr)
{
    MemoryObject * o = NULL;
    char * cptr = NULL;

    cptr = (char *)ptr;
    cptr -= sizeof(MemoryObject);
    o = (MemoryObject *)cptr;

    o->retainCount++;
}
```

## Reference Counting nach Jean-David Gadina[1]

Definition der *release*-Funktion:

```
void release(void * ptr)
{
    MemoryObject * o = NULL;
    char * cptr = NULL;

    cptr = (char *)ptr;
    cptr -= sizeof(MemoryObject);
    o = (MemoryObject *)cptr;

    if(--o->retainCount == 0)
    {
        free(o);
        o = NULL;
    }
}
```

## Reference Counting nach Jean-David Gadina

Pseudo Code für *testReferenceCounting*-Funktion:

```
unsigned long i = 0;
int * referencePointer = NULL;
int ** ptrs = (int **)calloc(numIterations, sizeof(int *));

for(i = 0; i < numIterations; ++i)
    ptrs[i] = (int *)alloc(sizeof(int));    // 1. Allocation

for(i = 0; i < numIterations; ++i) {
    referencePointer = ptrs[i];
    retain(referencePointer);               // 2. Retain
}

for(i = 0; i < numIterations; ++i)
    release(ptrs[i]);                       // 3. Release
free(ptrs);
```

## Reference Counting nach Jean-David Gadina

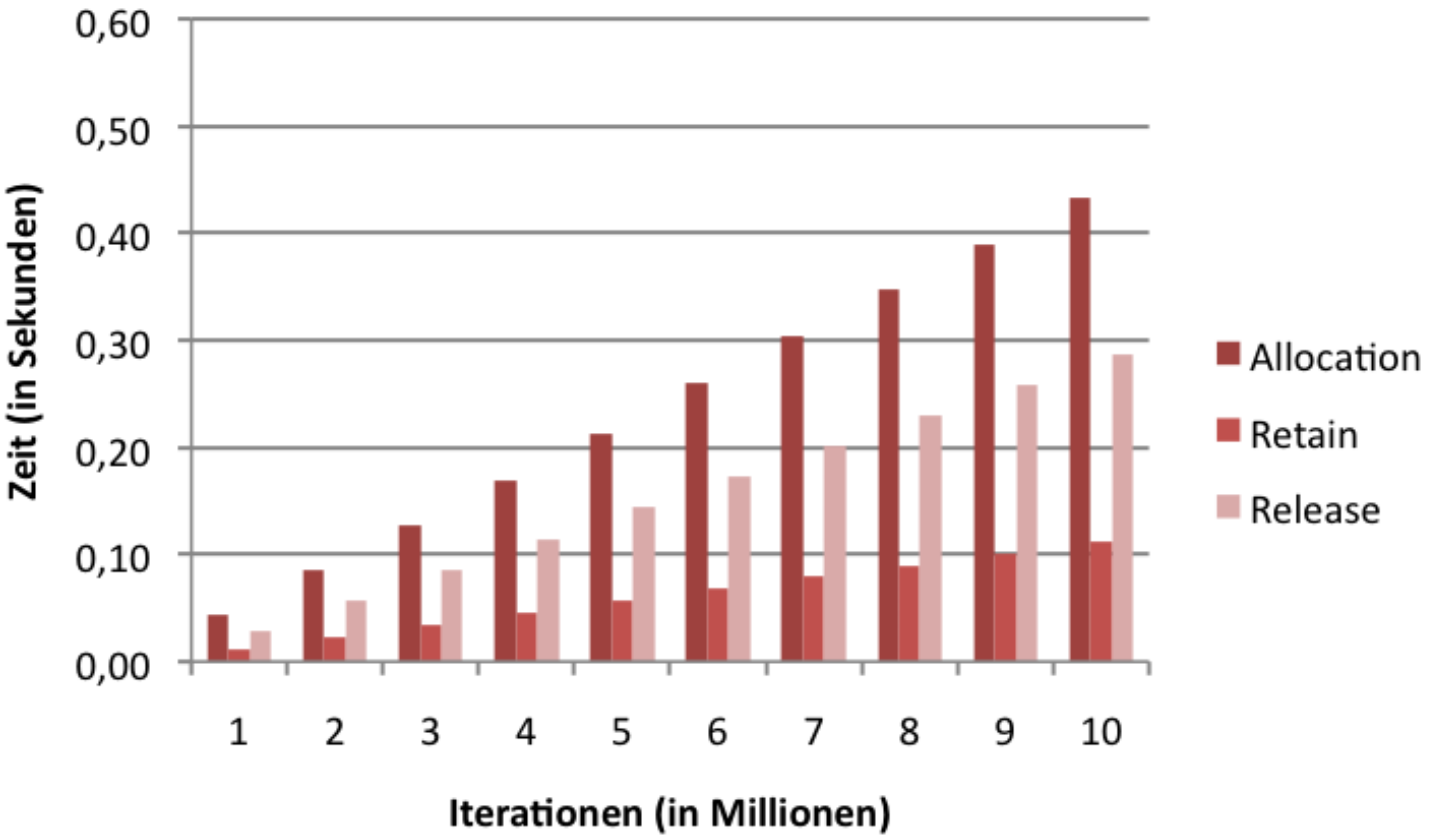


**Abbildung 5:** Performance: Reference Counting nach Jean-David Gadina.

# Reference Counting nach Jean-David Gadina

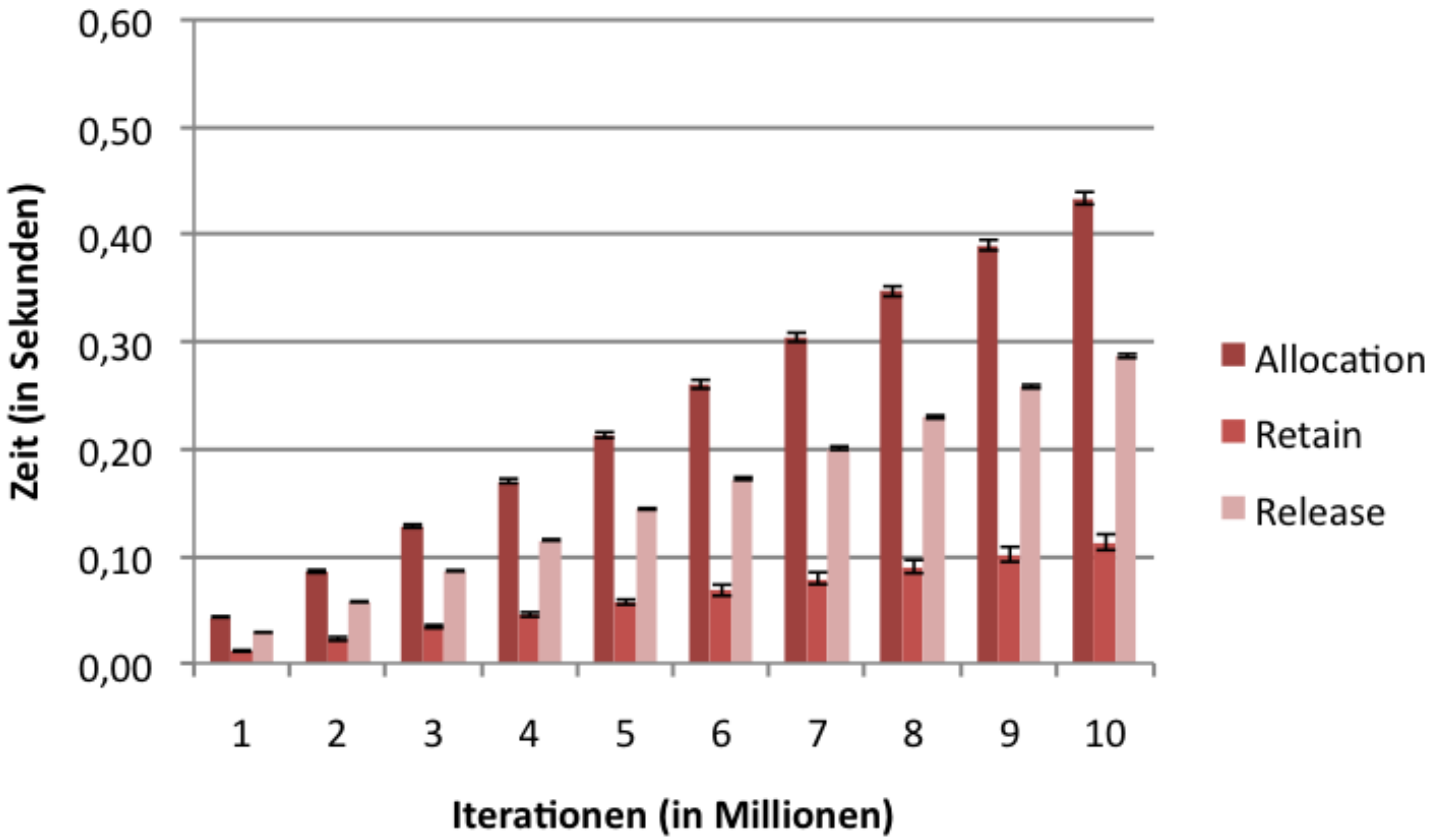


**Abbildung 5:** Performance: Reference Counting nach Jean-David Gadina.

## Reference Counting mit GLib

Pseudo Code für *testGLibReferenceCounting*-Funktion:

```
unsigned long i = 0;
gpointer referencePointer = NULL;
gpointer * ptrs = (gpointer *)calloc(numIterations,
                                     sizeof(gpointer));

for(i = 0; i < numIterations; ++i)
    ptrs[i] = my_object_new(0);              // 1. Allocation

for(i = 0; i < numIterations; ++i)           // 2. Retain
    referencePointer = g_object_ref(G_OBJECT(ptrs[i]));

for(i = 0; i < numIterations; ++i)
    g_object_unref(G_OBJECT(ptrs[i]));       // 3. Release

free(ptrs);
```

# Reference Counting mit GLib

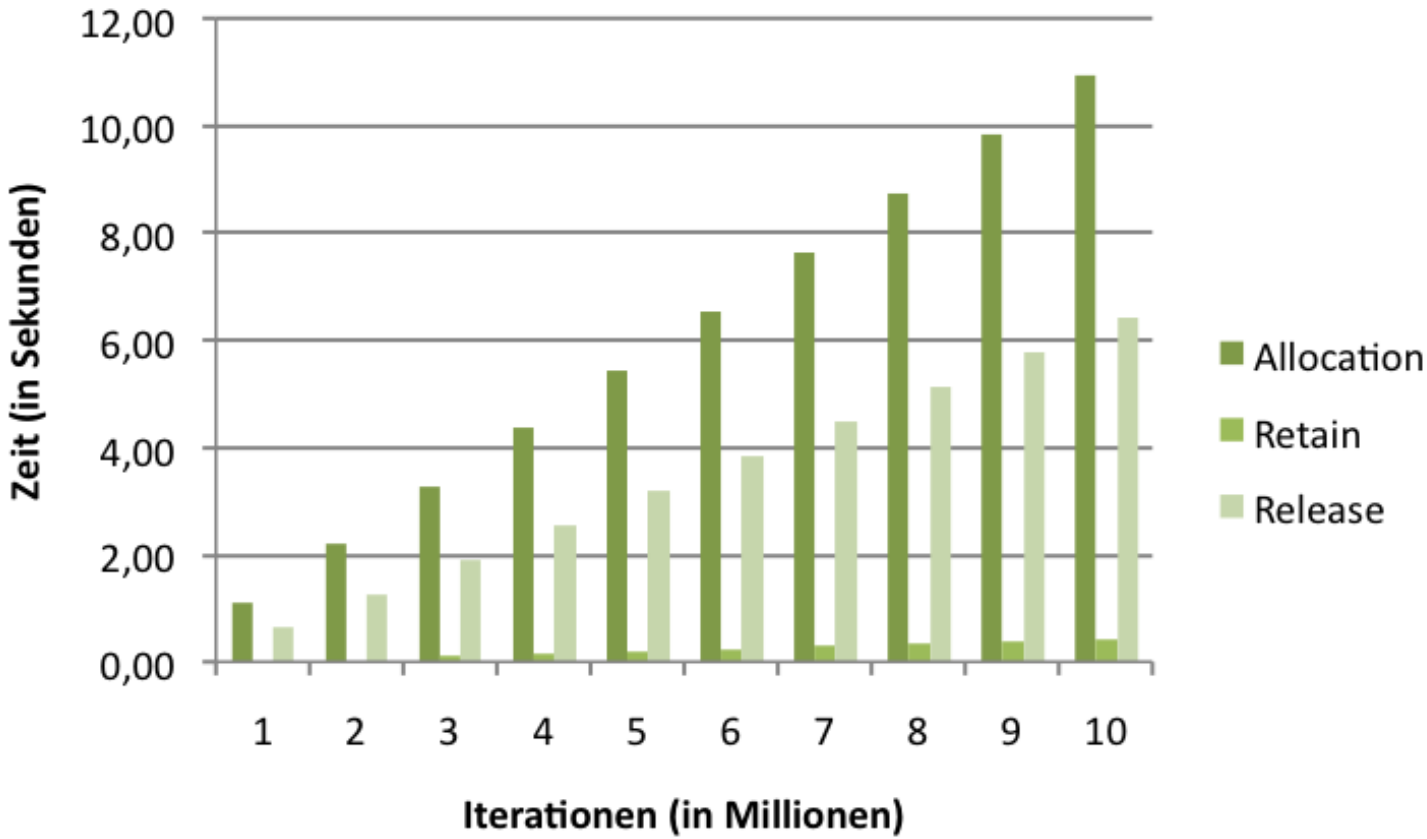


**Abbildung 6:** Performance: Reference Counting mit GLib.

# Reference Counting mit GLib

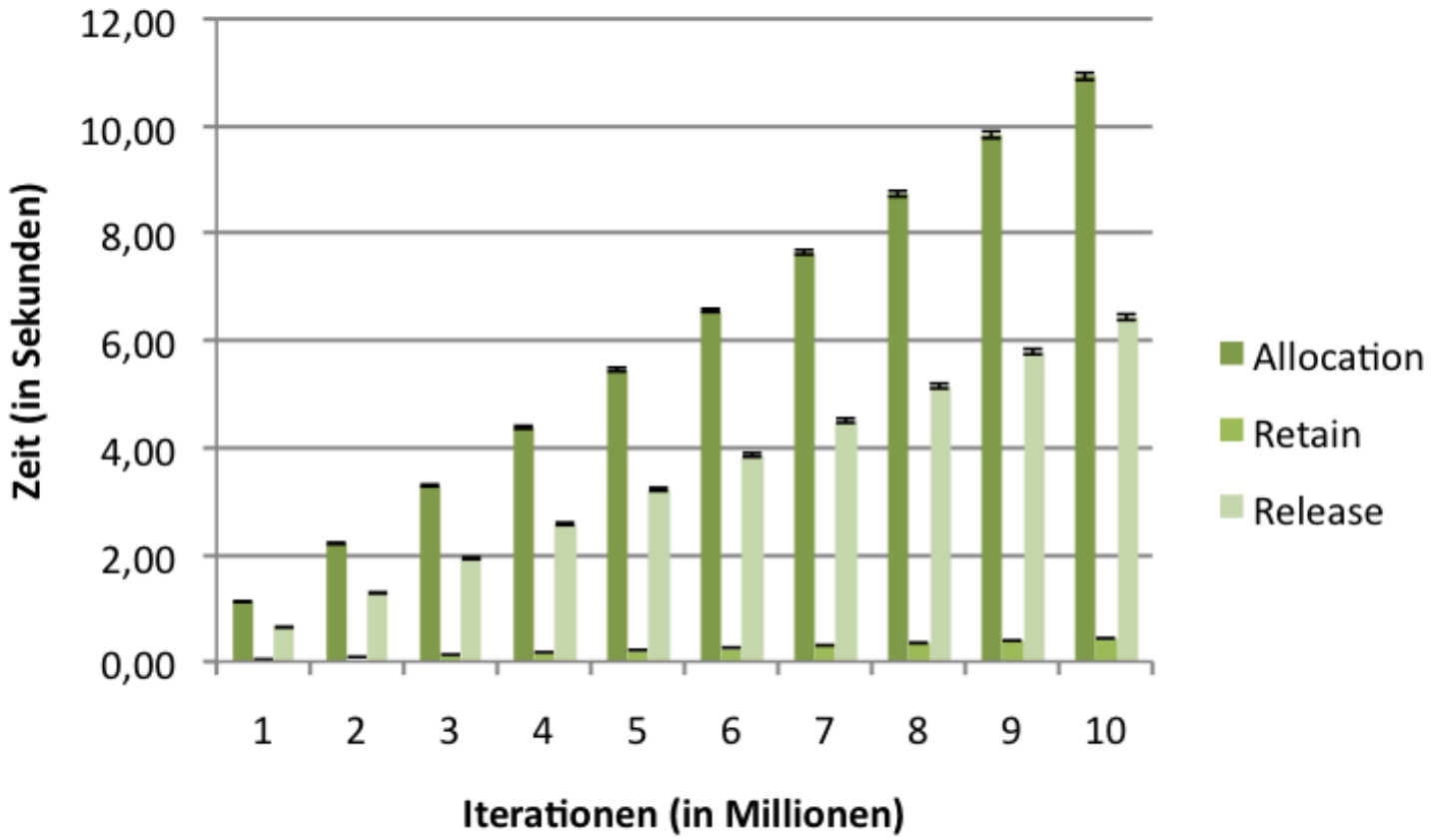


**Abbildung 6:** Performance: Reference Counting mit GLib.

# Evaluierung: Allocation Performancetest

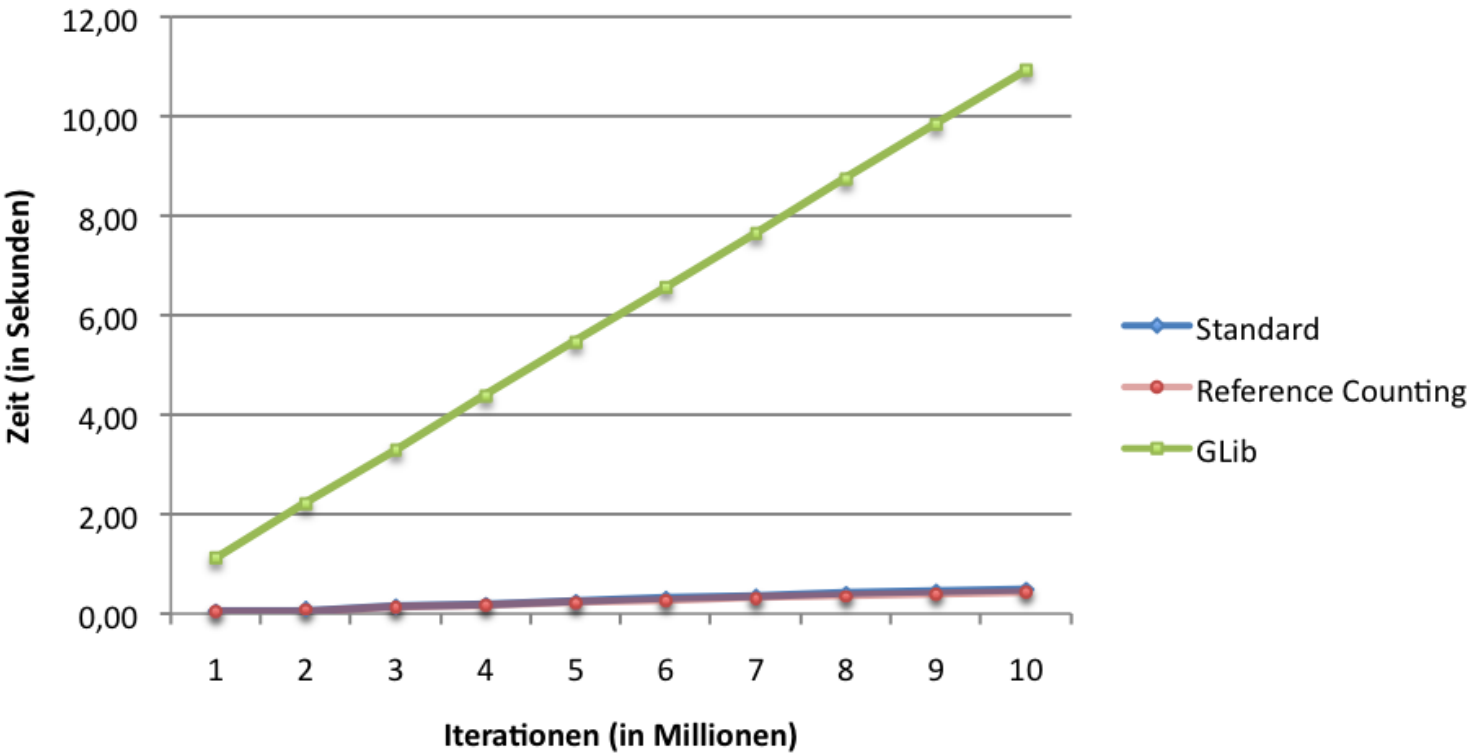


**Abbildung 7:** Performance: Allocation.

# Evaluierung: Allocation Performancetest

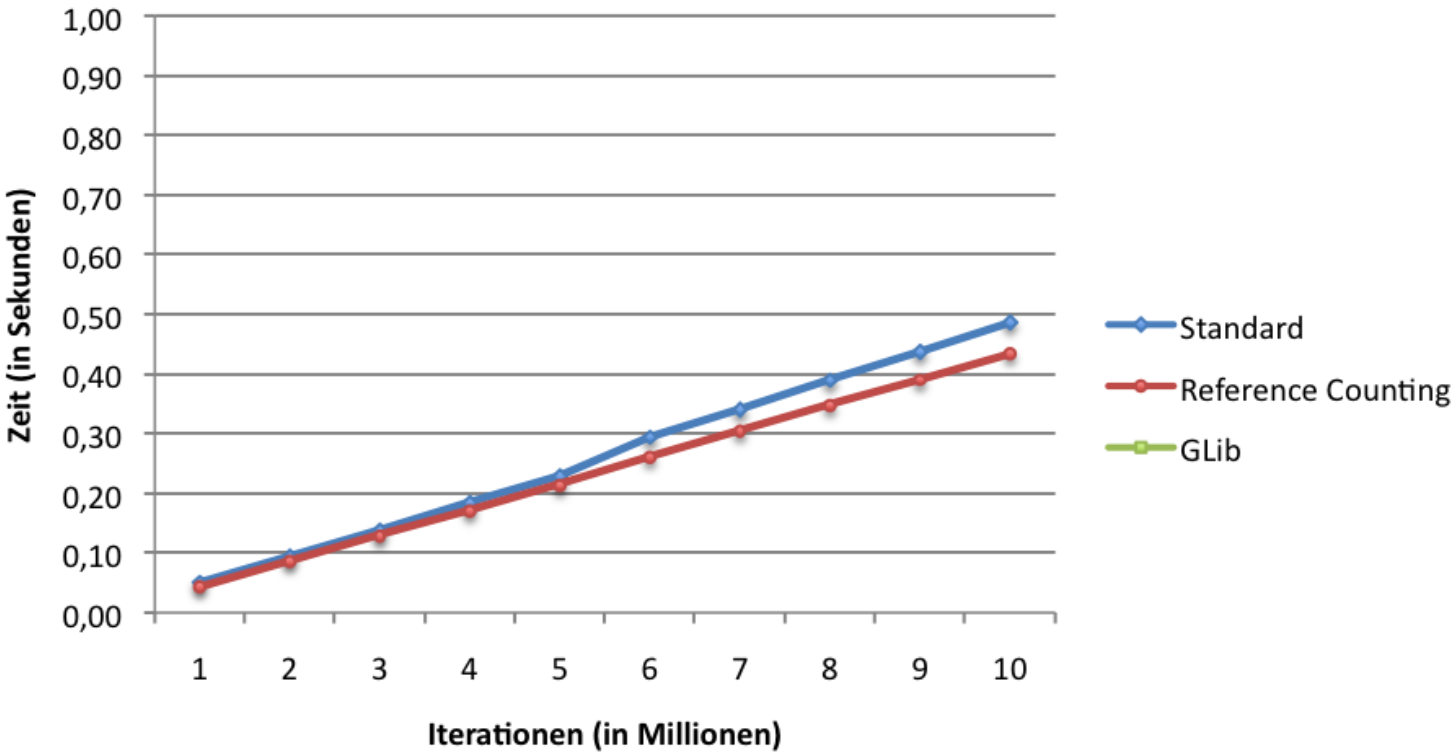


**Abbildung 7:** Performance: Allocation.

# Evaluierung: Retain Performancetest

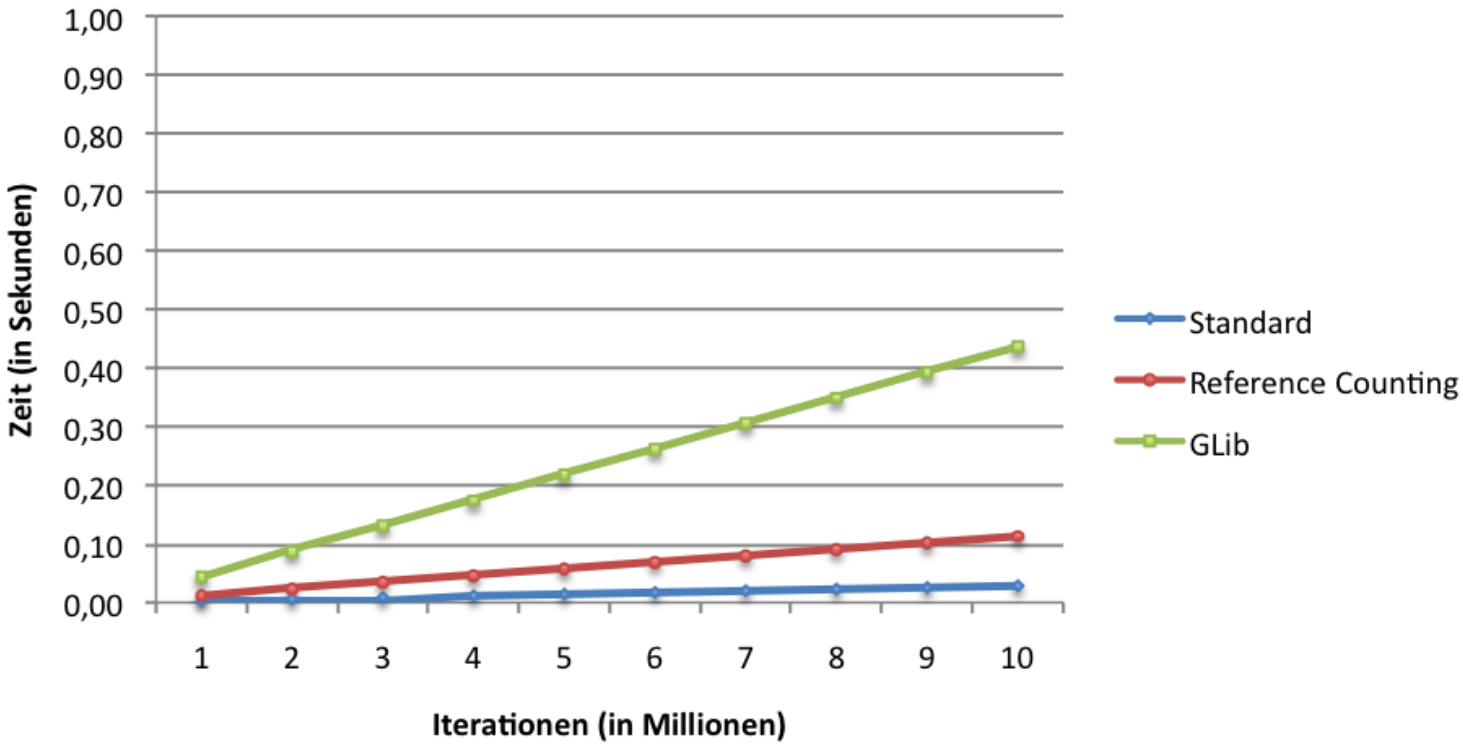


**Abbildung 8:** Performance: Retain.

# Evaluierung: Release Performancetest

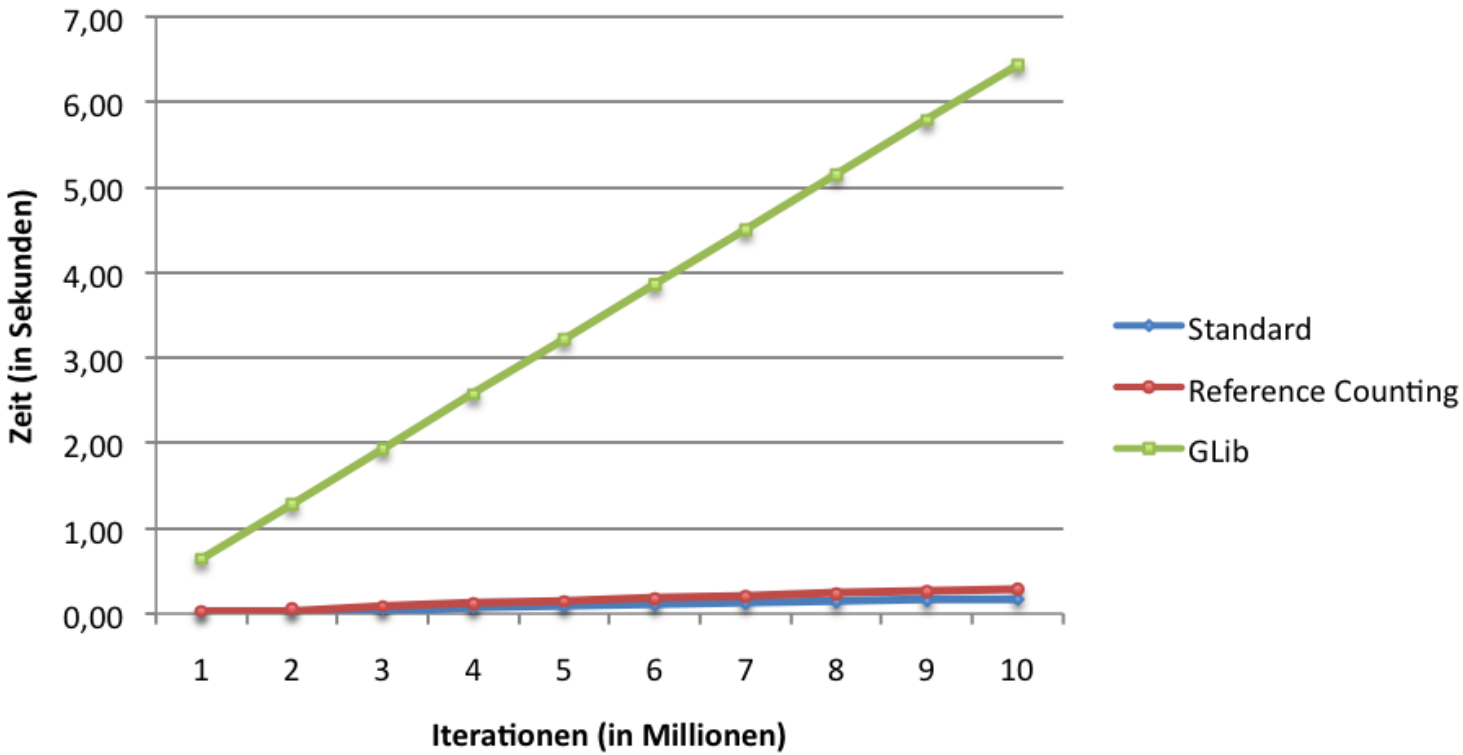


**Abbildung 9:** Performance: Release.

# Evaluierung: Release Performancetest

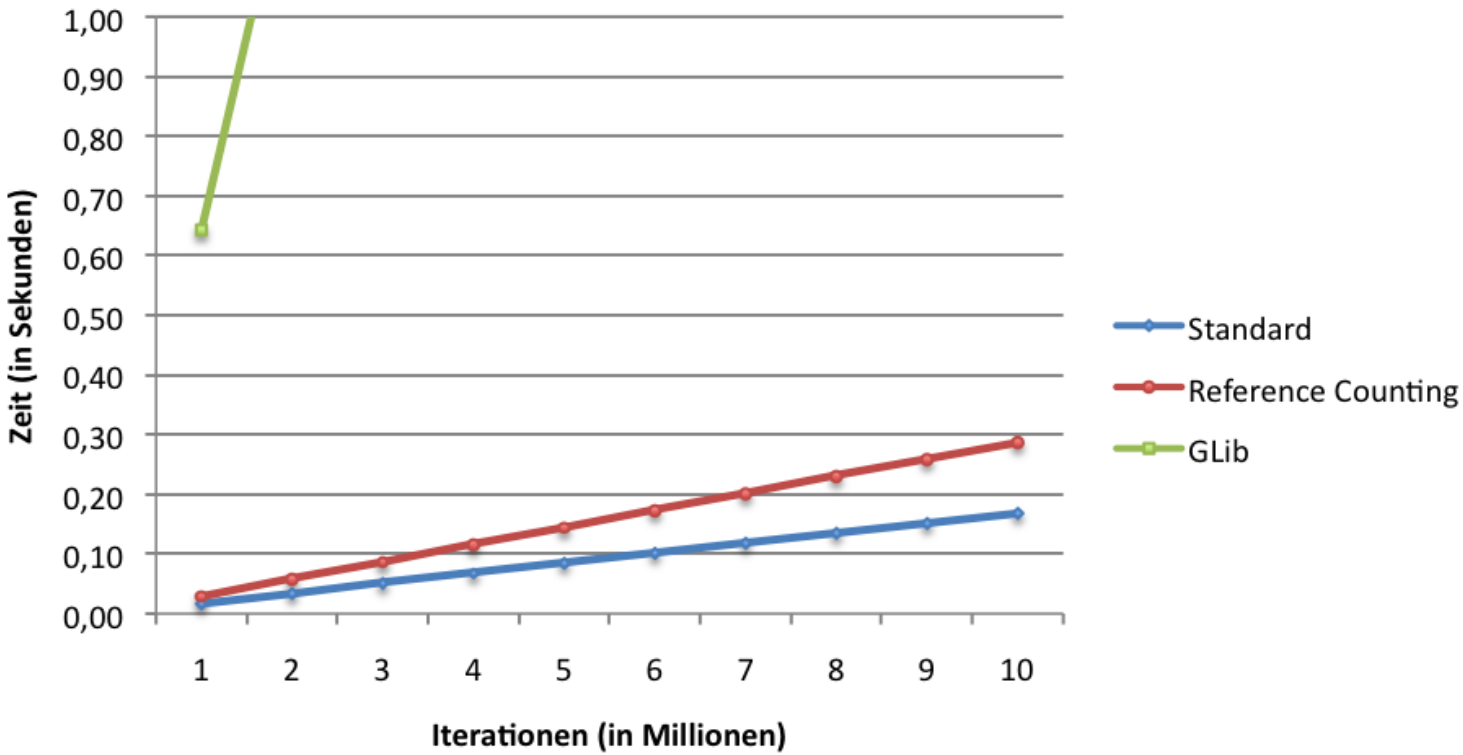


**Abbildung 9:** Performance: Release.

# Zusammenfassung

## Reference Counting

# Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.
  - **Echtzeitanwendungen**

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.
  - **Echtzeitanwendungen**
  - Systeme mit **limitierten Speicher**.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.
  - **Echtzeitanwendungen**
  - Systeme mit **limitierten Speicher**.
- Unterstützendes Konzept für **Entwurfsmuster**.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.
  - **Echtzeitanwendungen**
  - Systeme mit **limitierten Speicher**.
- Unterstützendes Konzept für **Entwurfsmuster**.
- **Keine Universallösung.**

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.
  - **Echtzeitanwendungen**
  - Systeme mit **limitierten Speicher**.
- Unterstützendes Konzept für **Entwurfsmuster**.
- **Keine Universallösung.**
  - **Memory Corruption** und **Memory Leaks** möglich.

## Zusammenfassung

**Reference Counting**:

- Weit **verbreitetes**, **unterstütztes** und zum Teil **integriertes** Konzept.
  - Implementierung in C möglich.
  - C Bibliotheken (z.B. *GLib*) frei verfügbar.
- **Effizientes Hilfsmittel** für Speichermanagement.
  - **Echtzeitanwendungen**
  - Systeme mit **limitierten Speicher**.
- Unterstützendes Konzept für **Entwurfsmuster**.
- **Keine Universallösung.**
  - **Memory Corruption** und **Memory Leaks** möglich.
  - **Zyklen**

## Literaturverzeichnis

[1] Jean-David Gadina.
Reference counting in ANSI-C.
*www.xs-labs.com*, 2013.

[2] Kevlin Henney.
C++ patterns-reference accounting.
In *in Proceedings of the EuroPLoP 2002 conference,(Irsee*.
Citeseer, 2002.

[3] Mac Developer Library.
About Memory Management.
*Advanced Memory Management Programming Guide*, 07 2012.

[4] Mac Developer Library.
Basic Memory Management Rules.
*Advanced Memory Management Programming Guide*, 07 2012.