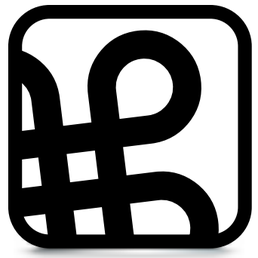


Macoun’11

Automatic Reference Counting

Daniel Höpfl

Was ist ARC?

ARC ist …

- … eine Art Speicher zu verwalten
- … eine Compilertechnologie

ARC ist …

- … eine Art Speicher zu verwalten
- … eine Compilertechnologie (mit Laufzeitkomponenten)
- … eine Erweiterung von Objective-C/C++

Was ist ARC nicht?

ARC ist nicht …

- … Garbage Collection
- … ohne den Entwickler möglich

Garbage Collection
vs.
Explizite Speicherverwaltung

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz, Emery D. Berger

http://www-cs.canisius.edu/~hertzm/gcmalloc-oopsla-2005.pdf

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz *
Computer Science Department
Canisius College
Buffalo, NY 14208
matthew.hertz@canisius.edu

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Garbage collection yields numerous software engineering benefits, but its quantitative impact on performance remains elusive. One can compare the cost of *conservative* garbage collection to explicit memory management in C/C++ programs by linking in an appropriate collector. This kind of direct comparison is not possible for languages designed for garbage collection (e.g., Java), because programs in these languages naturally do not contain calls to `free`. Thus, the actual gap between the time and space performance of

jar" memory manager eliminates the effects of consulting an oracle while measuring the costs of calling `malloc` and `free`. We evaluate two different oracles: a liveness-based oracle that aggressively frees objects immediately after their last use, and a reachability-based oracle that conservatively frees objects just after they are last reachable. These oracles span the range of possible placement of explicit deallocation calls.

We compare explicit memory management to both copying and non-copying garbage collectors across a range of benchmarks using the oracular memory manager, and present real (non-simulated) runs that lend further validity to our results. These results quantify the time-space tradeoff of garbage collection: with five times as much memory, an Appel-style generational collector with a non-copying mature space matches the performance of reachability-based explicit memory management. With only three times as much memory, the collector runs on average 17% slower than explicit memory management. However, with only twice as much memory, garbage collection degrades performance by nearly 70%. When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Dynamic storage management; D.3.4 [**Processors**]: Memory management (garbage collection)

General Terms

Experimentation, Measurement, Performance

Garbage collection, or automatic memory management, provides

the burden of memory management, eliminates most memory leaks, and improves modularity, while preventing accidental memory overwrites ("dangling pointers") [50, 59]. Because of these advantages, garbage collection has been incorporated as a feature of a number of mainstream programming languages.

Garbage collection can improve programmer productivity [48], but its impact on performance is difficult to quantify. Previous researchers have measured the runtime performance and space impact of *conservative*, non-copying garbage collection in C and C++ programs [19, 62]. For these programs, comparing the performance of explicit memory management to conservative garbage collection is a matter of linking in a library like the Boehm-Demers-Weiser collector [14]. Unfortunately, measuring the performance trade-off in languages designed for garbage collection is not so straightforward. Because programs written in these languages do not explicitly deallocate objects, one cannot simply replace garbage collection with an explicit memory manager. Extrapolating the results of studies with conservative collectors is impossible because precise, relocating garbage collectors (suitable only for garbage-collected languages) consistently outperform conservative, non-relocating garbage collectors [10, 12].

It is possible to measure the costs of garbage collection activity (e.g., tracing and copying) [10, 20, 30, 36, 56] but it is impossible to subtract garbage collection's effect on mutator performance. Garbage collection alters application behavior both by visiting and reorganizing memory. It also degrades locality, especially when physical memory is scarce [61]. Subtracting the costs of garbage collection also ignores the improved locality that explicit memory managers can provide by immediately recycling just-freed memory [53, 55, 57, 58]. For all these reasons, the costs of precise,

*Work performed at the University of Massachusetts Amherst.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

"with five times as much memory, an Appel-style generational collector with non-copy mature space matches the performance of reachability-based explicit memory management."

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz *
Computer Science Department
Canisius College
Buffalo, NY 14208
matthew.hertz@canisius.edu

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Garbage collection yields numerous software engineering benefits, but its quantitative impact on performance remains elusive. One can compare the cost of *conservative* garbage collection to explicit memory management in C/C++ programs by linking in an appropriate collector. This kind of direct comparison is not possible for languages designed for garbage collection (e.g., Java), because programs in these languages naturally do not contain calls to `free`. Thus, the actual gap between the time and space performance of explicit memory management and *precise*, copying garbage collection remains unknown.

We compare explicit memory management to both copying and non-copying garbage collectors across a range of benchmarks using the oracular memory manager, and present real (non-simulated) runs that lend further validity to our results. These results quantify the time-space tradeoff of garbage collection: with five times as much memory, an Appel-style generational collector with a non-copying mature space matches the performance of reachability-based explicit memory management. With only three times as much memory, the collector runs on average 17% slower than explicit memory management. However, with only twice as much memory, garbage collection degrades performance by nearly 70%. When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Dynamic storage management; D.3.4 [**Processors**]: Memory management (garbage collection)

General Terms

Experimentation, Measurement, Performance

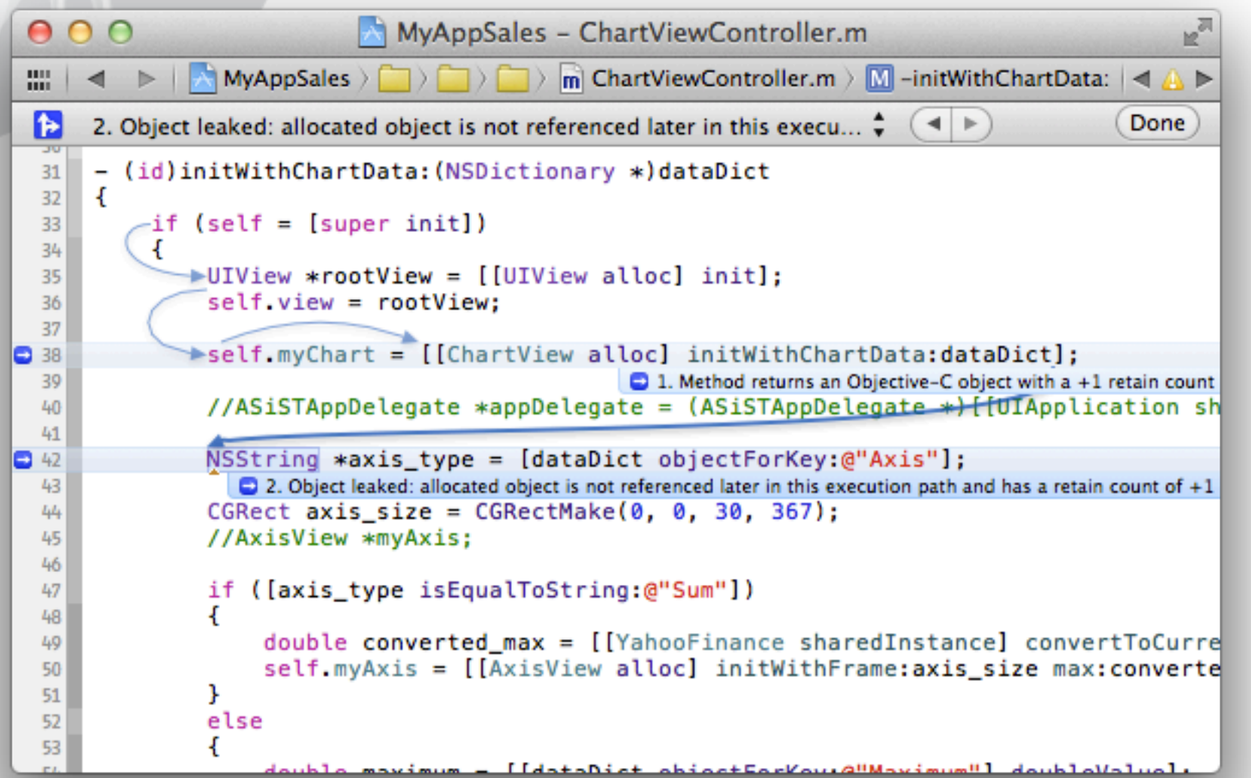
> “However, with only twice as much memory, garbage collection degrades performance by nearly 70%.”

*Work performed at the University of Massachusetts Amherst.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

Wie funktioniert ARC?

Static Analyzer



Neue Regeln

- Aufruf verboten: `retain`, `release`, `autorelease`, `dealloc`, `retainCount`
- Anstelle `NSAutoreleasePool` muss `@autoreleasepool` benutzt werden
- Keine Objective-C-Pointer in `structs`
- Casts zwischen Objective-C-Pointer und C-Pointer müssen „dem Compiler erläutert“ werden

Neue Qualifier

- Cast: `__bridge, __bridge_retained, __bridge_transfer`
- Property: `strong (retain), weak, unsafe_unretained (assign)`
- Qualifier: `__strong, __autoreleasing, __weak, __unsave_unretained`

Naming Conventions

- Erzeuger: `alloc…, copy…, mutableCopy…, new…`
- Konstruktoren: `init…`
- Keine Property: `alloc…, copy…, mutableCopy…, new…, init…`

Pointer

- Alle Pointer werden mit `nil` initialisiert

Beispiele

Beispiele

```
{
    NSString *str = [[NSString alloc]
                          initWithCString:”Macoun”
                                 encoding:NSUTF8StringEncoding];

    NSLog(@”Benutzen ... %@”, str);

    [str release];
}
```

Beispiele

```
{
    __strong NSString *str = [[NSString alloc]
                              initWithCString:”Macoun”
                                     encoding:NSUTF8StringEncoding];

    NSLog(@”Benutzen ... %@”, str);
}
```

Beispiele

```
{
    NSString *str = [[NSString alloc]
                          initWithCString:”Macoun”
                                 encoding:NSUTF8StringEncoding];

    NSLog(@”Benutzen ... %@”, str);
}
```

Beispiele

```
@interface ... {
   NSString *_name;
}

...

- (void) setName:(NSString *)name {
   NSString *tmpName = _name;
   _name = [name retain];
   [tmpName release];
}
```

Beispiele

```
@interface ... {
   NSString *_name;
}

...

- (void) setName:(NSString *)name {
   _name = name;
}
```

Beispiele

```
- (void) dealloc {
    [_name release];

    [[NSNotificationCenter defaultCenter] removeObserver:self];

    [super dealloc];
}
```

Beispiele

```
- (void) dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

Beispiele

```
int main (int argc, const char *argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    int retVal = UIApplicationMain(argc, argv, nil, nil);

    [pool release];

    return retVal;
}
```

Beispiele

```
int main (int argc, const char *argv[]) {

    @autoreleasepool {
        int retVal = UIApplicationMain(argc, argv, nil, nil);
        return retVal;
    }

}
```

Beispiele

```
struct {
    NSString *date;
    NSUInteger i;
} debugdata[] = {
    {@"01.01.2010 10:45", 5},
    {@"02.01.2010 10:45", 23},
    {@"03.01.2010 10:45", 42},
    ...
};
```

Beispiele

```
struct {
    __unsafe_unretained NSString *date;
    NSUInteger i;
} debugdata[] = {
    {@"01.01.2010 10:45", 5},
    {@"02.01.2010 10:45", 23},
    {@"03.01.2010 10:45", 42},
    ...
};
```

Beispiele

```
NSDictionary *animInfo = [NSDictionary dictionaryWith...];
// ...
[UIView beginAnimations:@”my Animation”
                context:(void *)[animInfo retain]];
```

```
/*
  + (void) beginAnimations:(NSString *)
                   context:(void *)context;
*/
```

Beispiele

```
NSDictionary *animInfo = [NSDictionary dictionaryWith...];
// ...
[UIView beginAnimations:@”my Animation”
                context:(__bridge_retained void *)animInfo];
```

```
/*
  + (void) beginAnimations:(NSString *)
                   context:(void *)context;
*/
```

Beispiele

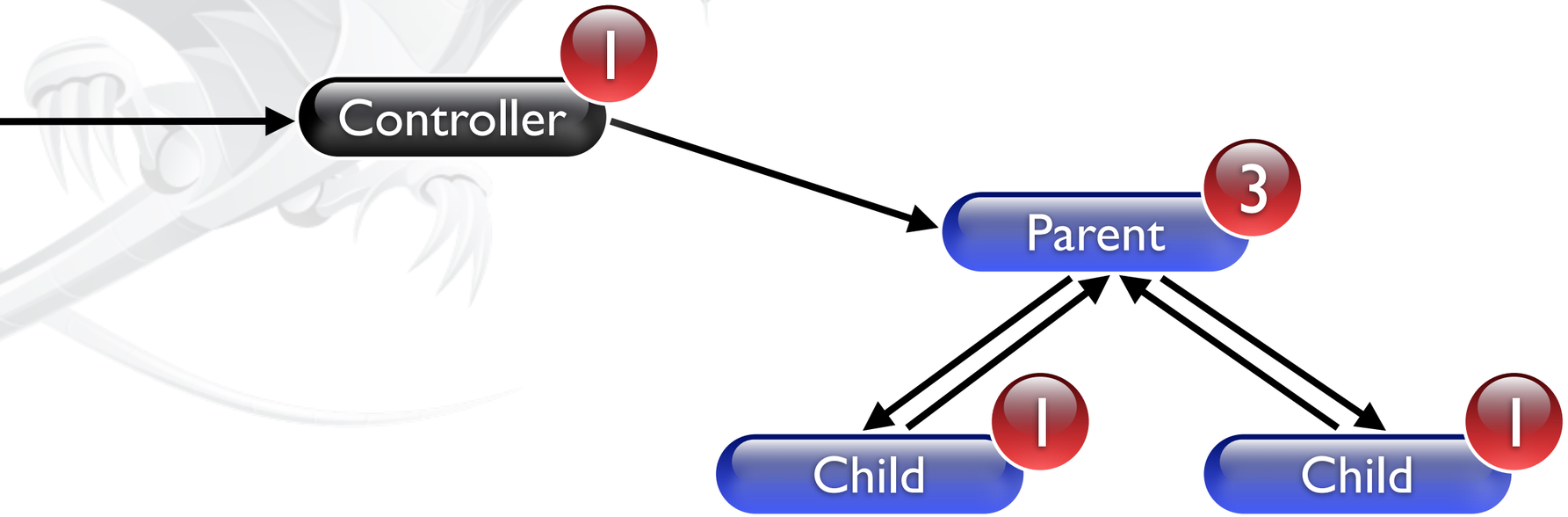
```
 ...
- (void) myAnimationDone:(NSString *)animationId
                finished:(NSNumber *)finished
                 context:(void *)context {
   if ([finished boolValue]) {
      NSDictionary *animInfo = (NSDictionary *)context;
      UIView *view = (UIView *) [animInfo objectForKey:...];
      [view removeFromSuperview];
      [animInfo release];
   }
}
```

Beispiele

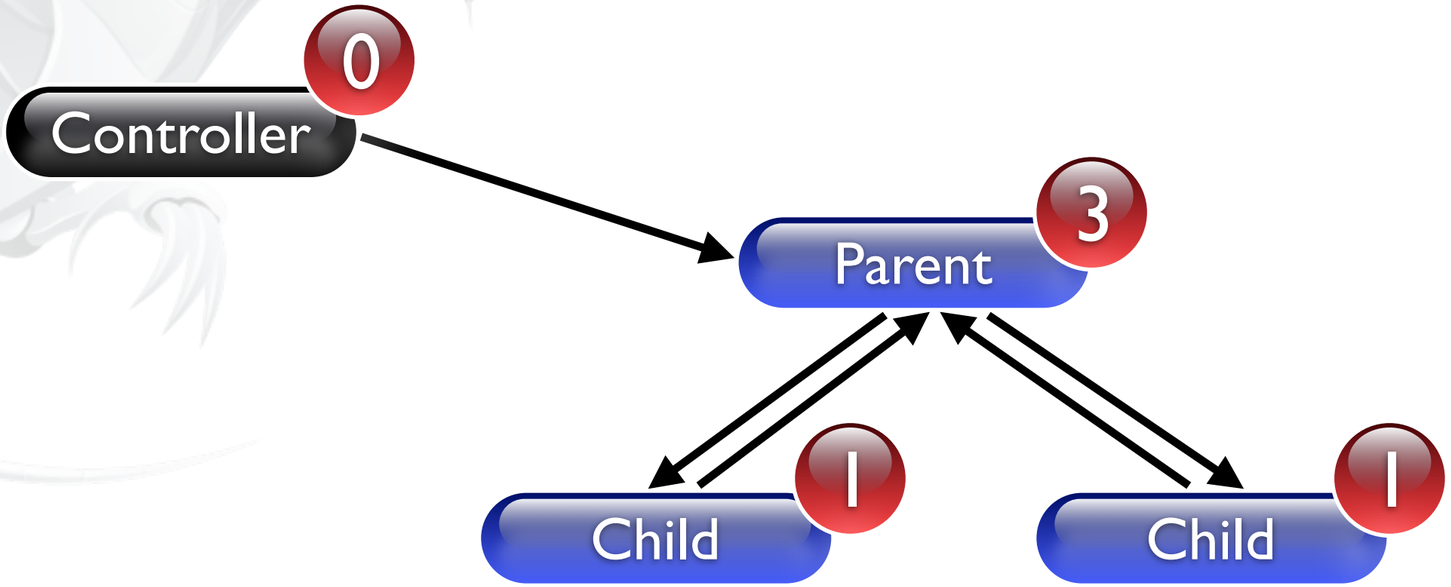
```
 ...
- (void) myAnimationDone:(NSString *)animationId
                finished:(NSNumber *)finished
                 context:(void *)context {
   if ([finished boolValue]) {
      NSDictionary *animInfo =
                (__bridge_transfer NSDictionary *)context;
      UIView *view = (UIView *) [animInfo objectForKey:...];
      [view removeFromSuperview];
   }
}
```

Strong & Weak

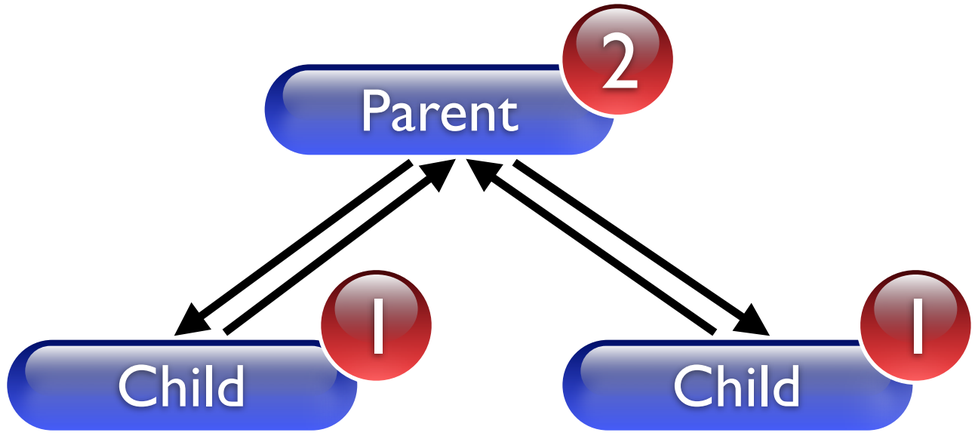
Strong



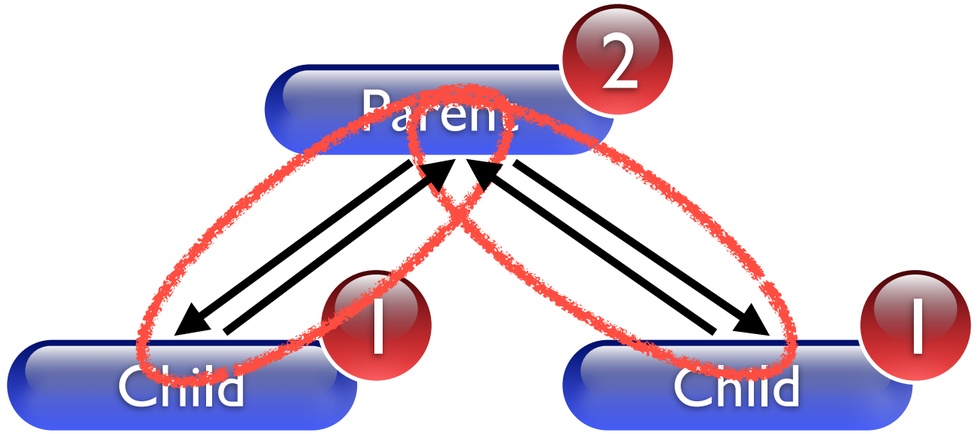
Strong



Strong



Strong



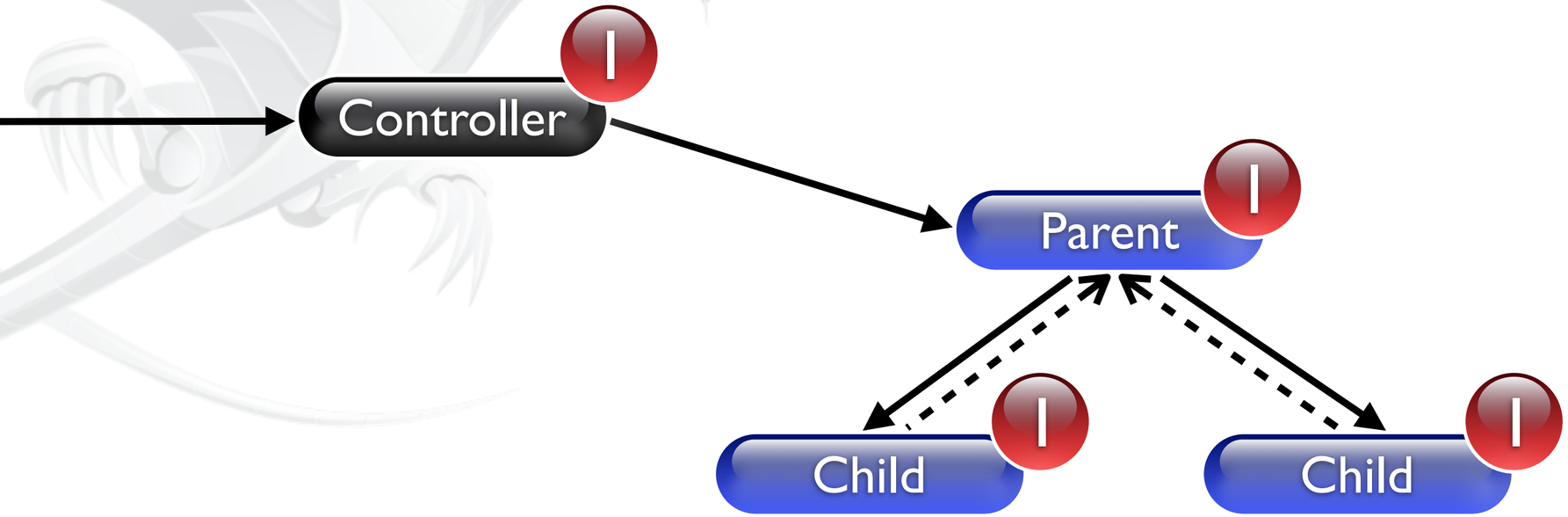
Weak-Pointer

- Erhalten das Ziel-Objekt nicht am Leben
- Werden auf `nil` gesetzt, wenn das Ziel freigegeben wird
- Verlangen Laufzeitunterstützung (MacOS X ≥10.7, iOS ≥5.0)
  - Ältere Systeme: `unsafe_unretained`

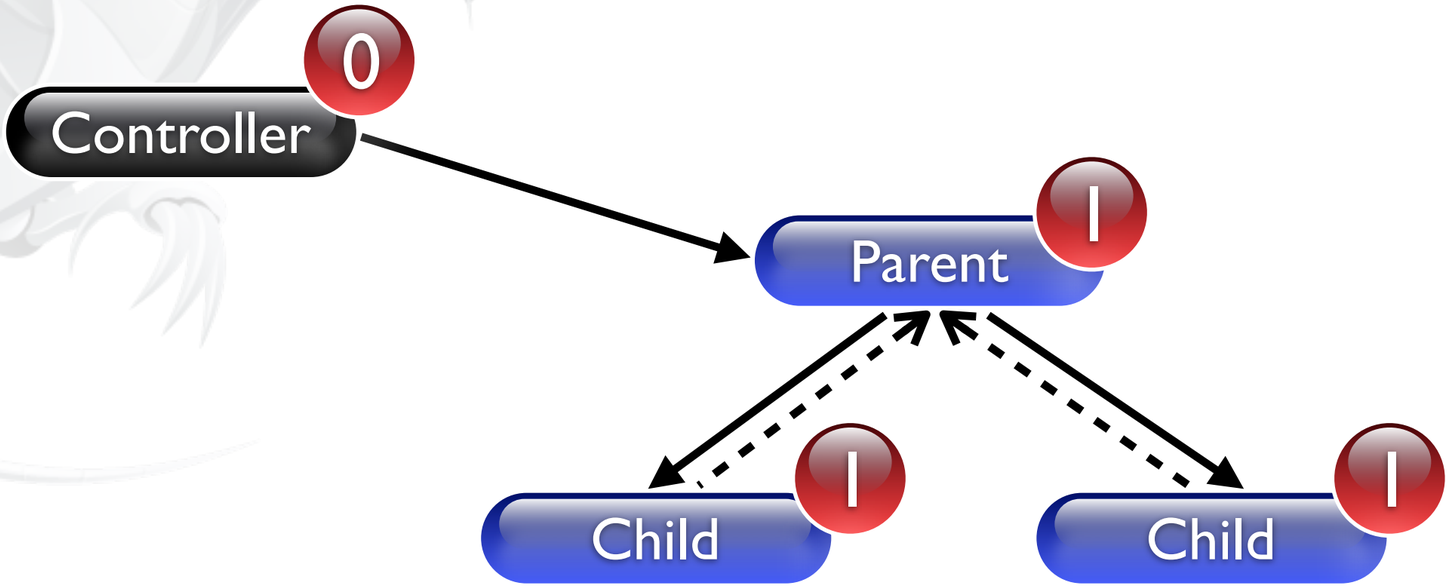
Weak-Pointer

- Empfohlen wenn Deployment Target: MacOS X ≥10.7, iOS ≥5.0
- Vorsicht: Nicht von allen Klassen unterstützt! (`NSWindow, …`)
- `- (BOOL) allowsWeakReference;`

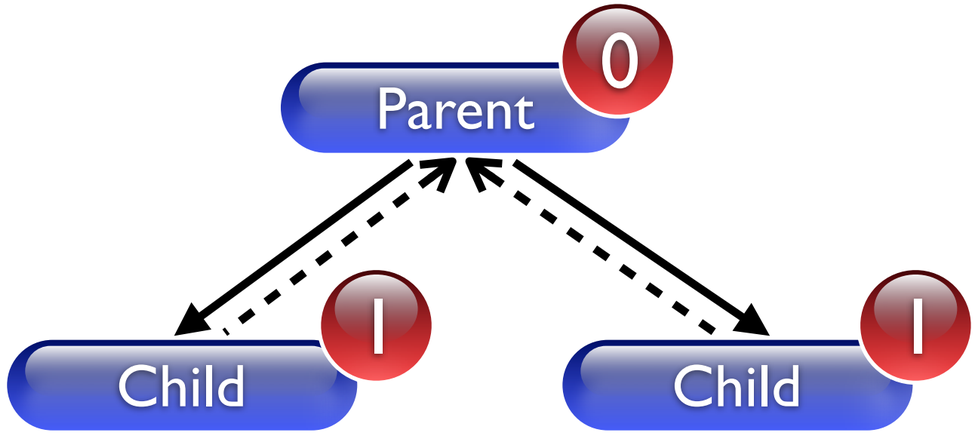
Weak-Pointer



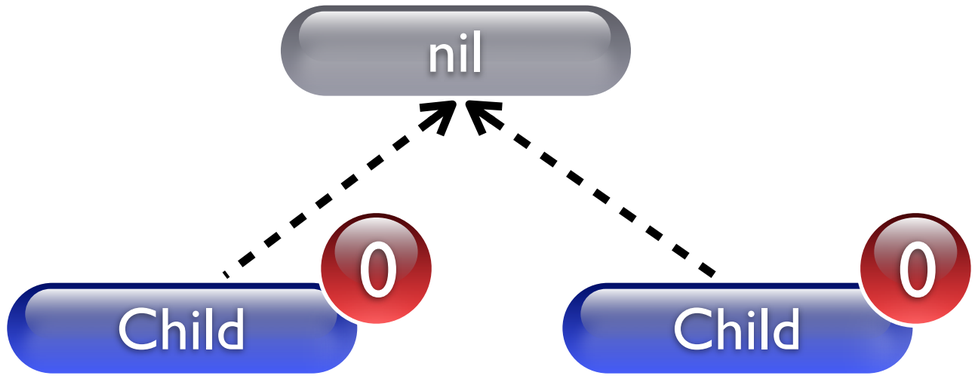
Weak-Pointer



Weak-Pointer



Weak-Pointer



Weak-Pointer

nil

Eltern **halten** ihre Kinder

Kinder **kennen** ihre Eltern

DEMO

Sonstiges

Core Foundation

- Core Foundation: Kein Objective-C, damit nicht von ARC verwaltet.
- Unterstützung für die Legacy-Methoden, z.B.

```
UIColor: - (CGColorRef) CGColor;
```

Collections

- Alle Collections (`NSArray`, `NSSet`, …) halten die Einträge `__strong`
- Gilt auch für `NSHashTable` und `NSMapTable`

Objective-C++

- Wird voll unterstützt!
- Bei STL-Containern ist die Angabe von `__strong` notwendig:

```
std::vector<__strong id> liste;
```

Out-Parameter

- Häufigster Anwendungsfall: `NSError **`
- Typische Anwendung:

```
- (void) doSomethingWithError:(NSError * __autoreleasing *)error;
```

Out-Parameter

```
- (void) doWithError:(NSError * __autoreleasing *)error {...}
```

```
{
   NSError *error;

   [obj doWithError:&error];
   if (error) {
      return;
   }

   NSLog(@"OK");
}
```

Out-Parameter

```
- (void) doWithError:(NSError * __autoreleasing *)error {...}
```

```
{
   __strong NSError *error;

   [obj doWithError:(NSError * __autoreleasing *) &error];
   if (error) {
      return;
   }

   NSLog(@"OK");
}
```

Out-Parameter

```
- (void) doWithError:(NSError * __autoreleasing *)error {...}
```

```
{
   __strong NSError *error;

   __autoreleasing NSError *tmp = error;
   [obj doWithError:&tmp];
   error = tmp;
   if (error) {
      return;
   }
// ...
```

Out-Parameter

```
- (void) doWithError:(NSError * __autoreleasing *)error {...}
```

```
{
   __autoreleasing NSError *error;

   [obj doWithError:&error];
   if (error) {
      return;
   }

   NSLog(@"OK");
}
```

Out-Parameter

```
- (void) doWithError:(NSError * __strong *)error {...}
```

```
{
   NSError *error;

   [obj doWithError:&error];
   if (error) {
      return;
   }

   NSLog(@"OK");
}
```

Blocks

- Speicherverwaltung anfällig für Fehler:

```
return [[^{...} copy] autorelease];
self.handler = [^{...} copy];
static id handler = [^{...} copy];
```

- Unter ARC sehr einfach:

```
return ^{...};
self.handler = ^{...};
static id handler = ^{...};
```

Blocks

- Neue Falle: Retain Cycles
  - Zugriffe auf `ivars` fangen `self`
  - `__block` nicht mehr `__unsafe_unretained` sondern `__strong`

Blocks

```
@synthesize (strong) mark = _mark;
@synthesize (strong) service = _service;
```

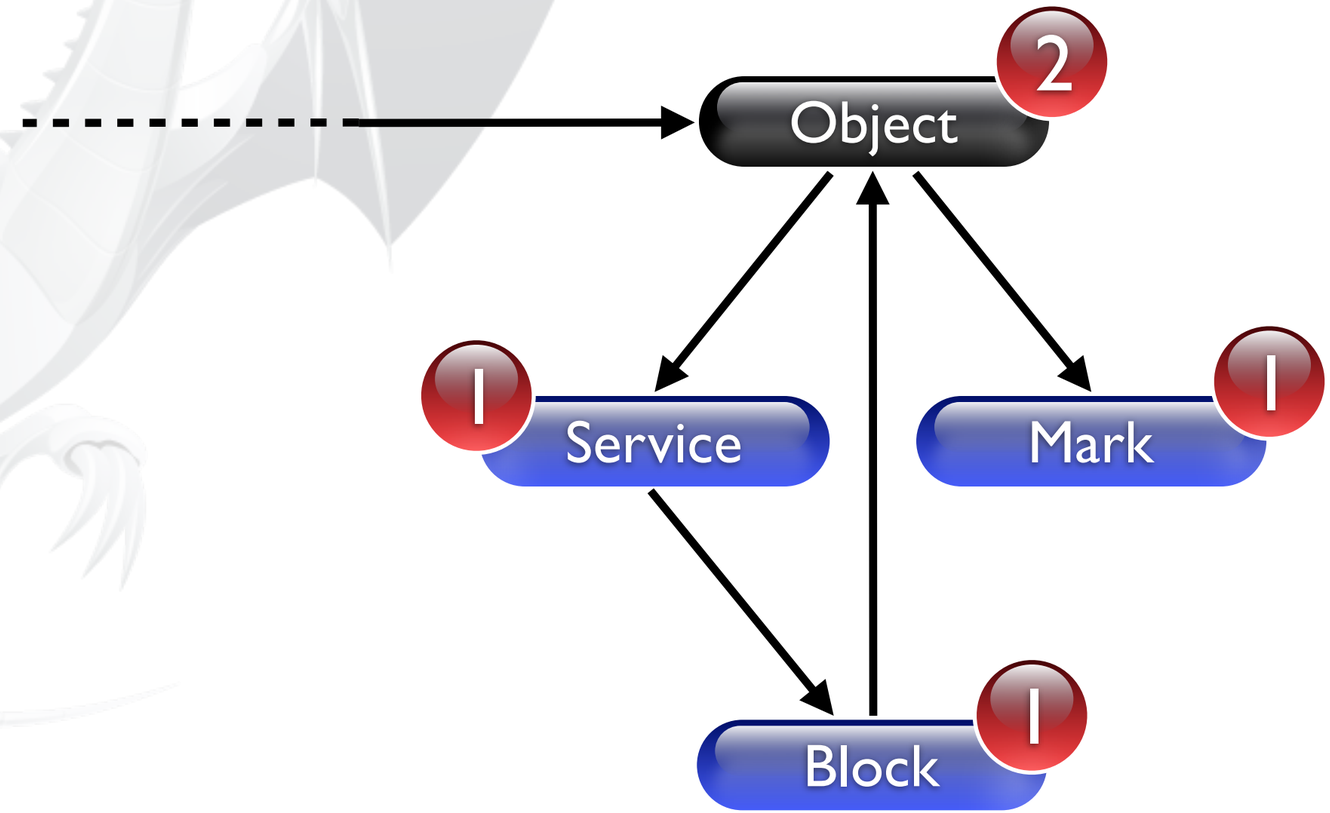
```
- (id) init {
// ...
  _service = [LogfileMarker logfileMarkerWithBlock:^{
                NSLog(@"%@", _mark);
              }];
// ...
}
```

Blocks

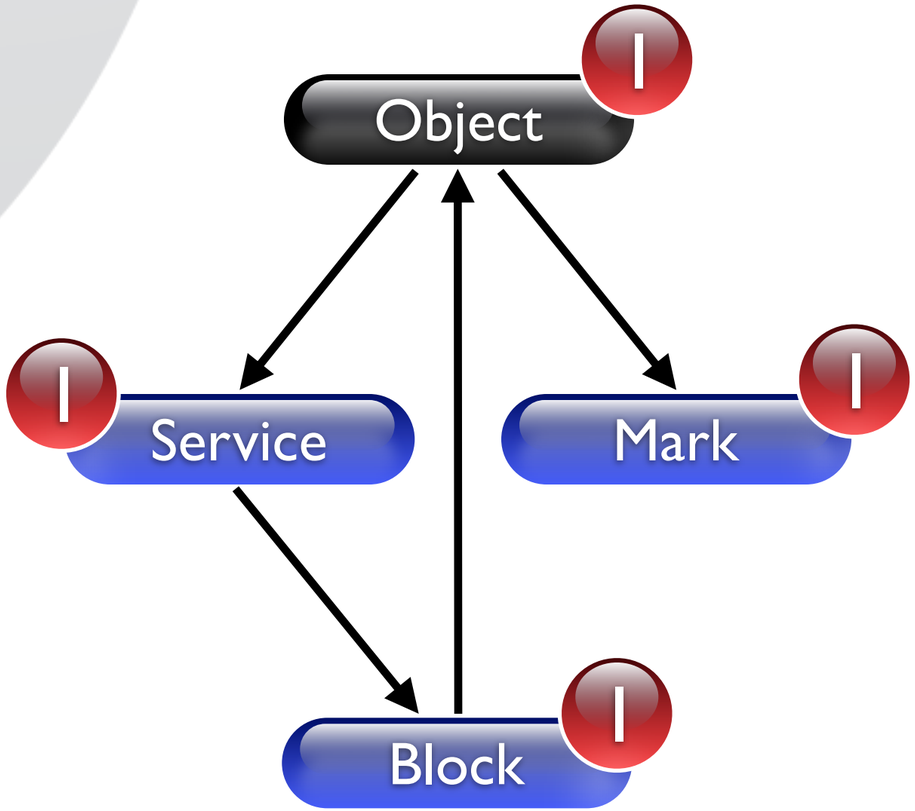
```
@synthesize (strong) mark = _mark;
@synthesize (strong) service = _service;
```

```
- (id) init {
// ...
  _service = [LogfileMarker logfileMarkerWithBlock:^{
                NSLog(@"%@", self->_mark);
              }];
// ...
}
```

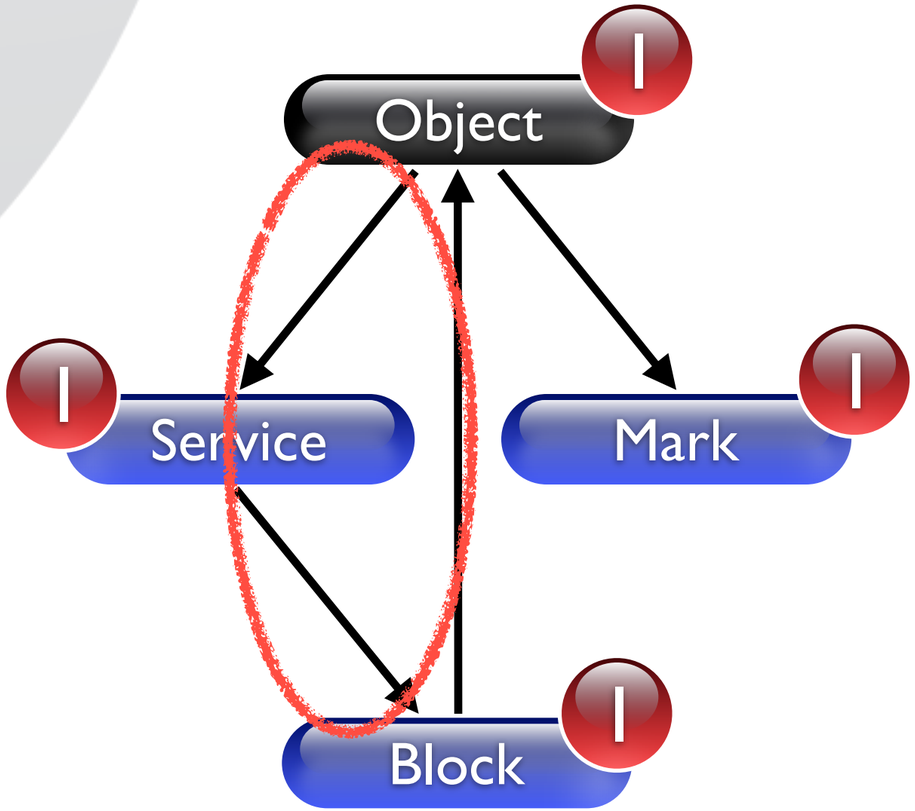
Blocks



Blocks



Blocks

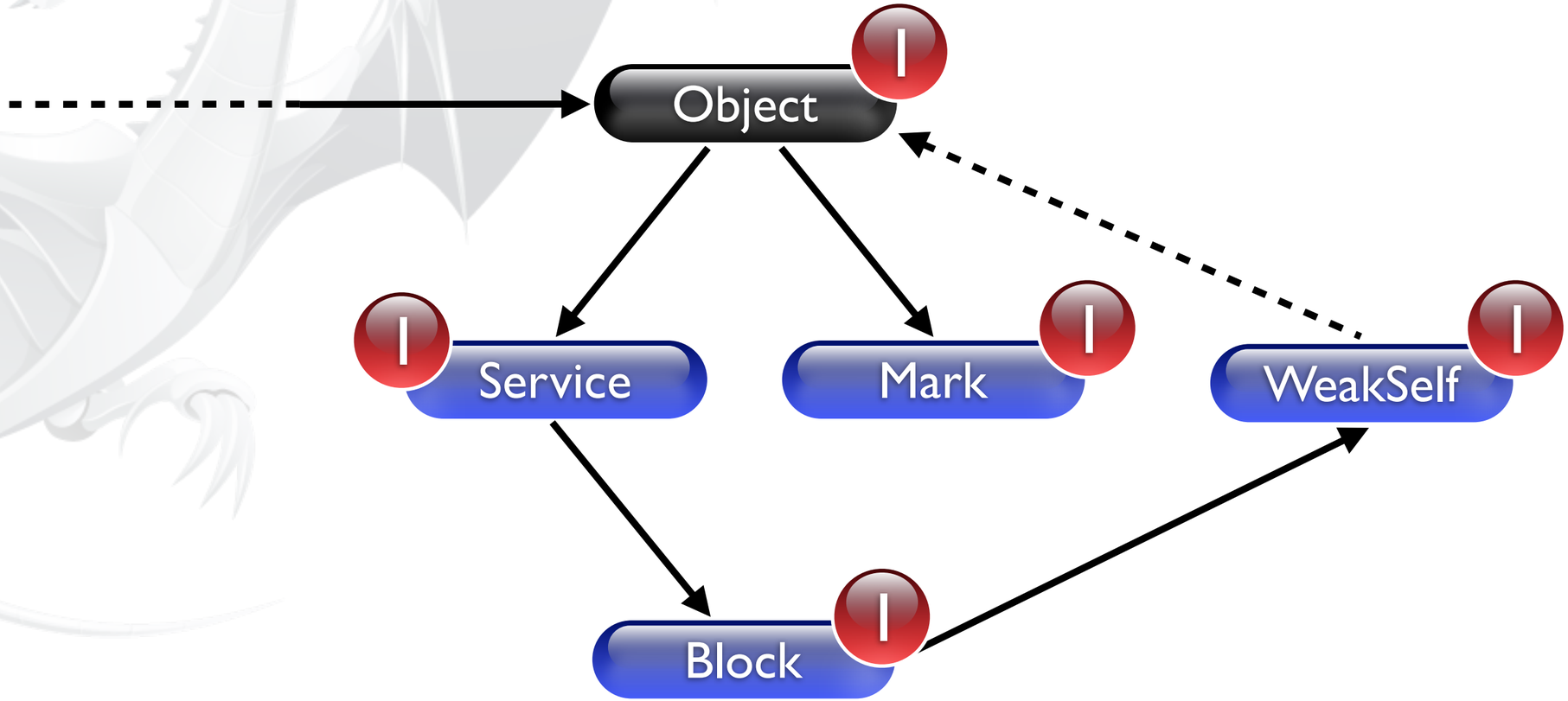


Blocks

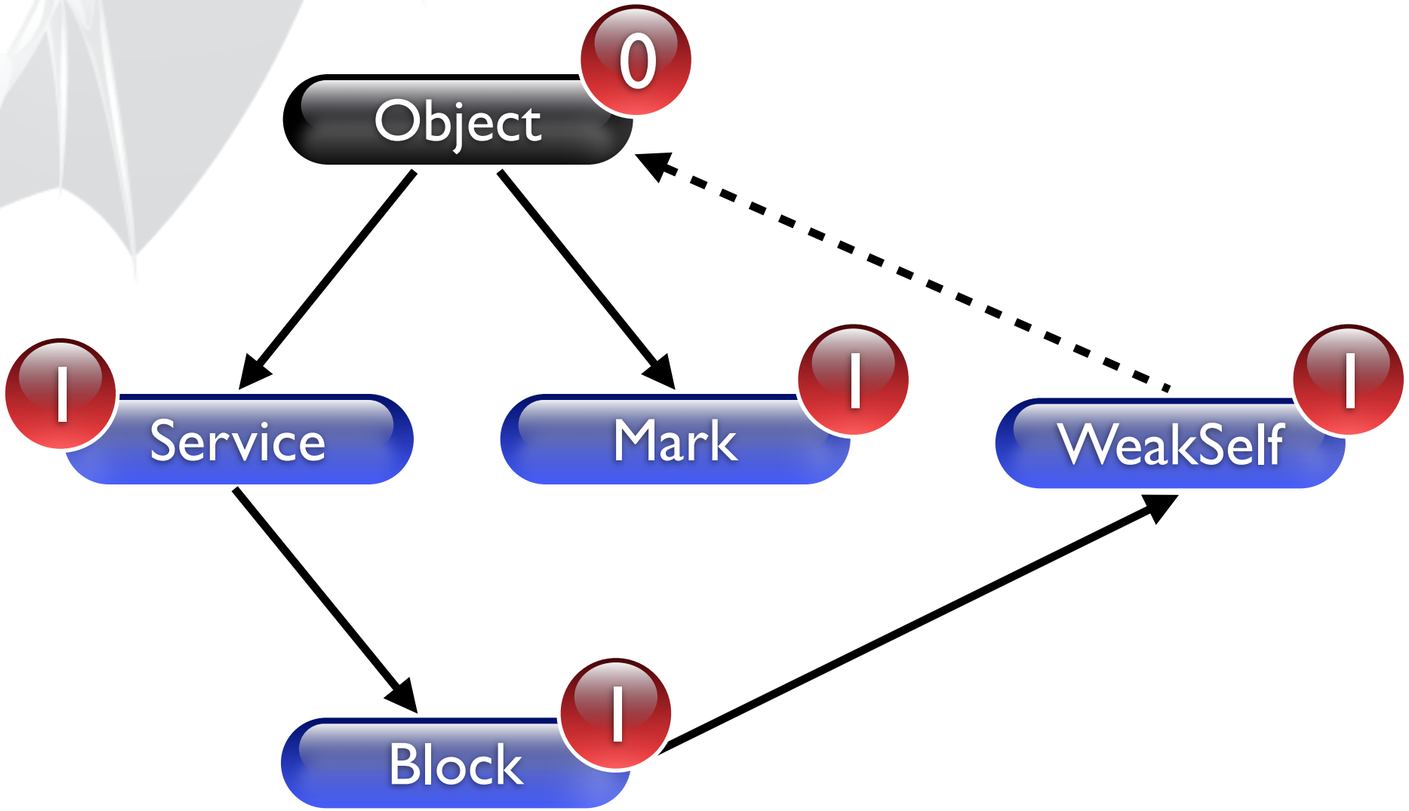
```
@synthesize (strong) mark = _mark;
@synthesize (strong) service = _service;
```

```
- (id) init {
// ...
  __weak id weakSelf = self;
  _service = [LogfileMarker logfileMarkerWithBlock:^{
                id strongSelf = weakSelf;
                if (strongSelf)
                  NSLog(@"%@", strongSelf->_mark);
              }];
// ...
}
```

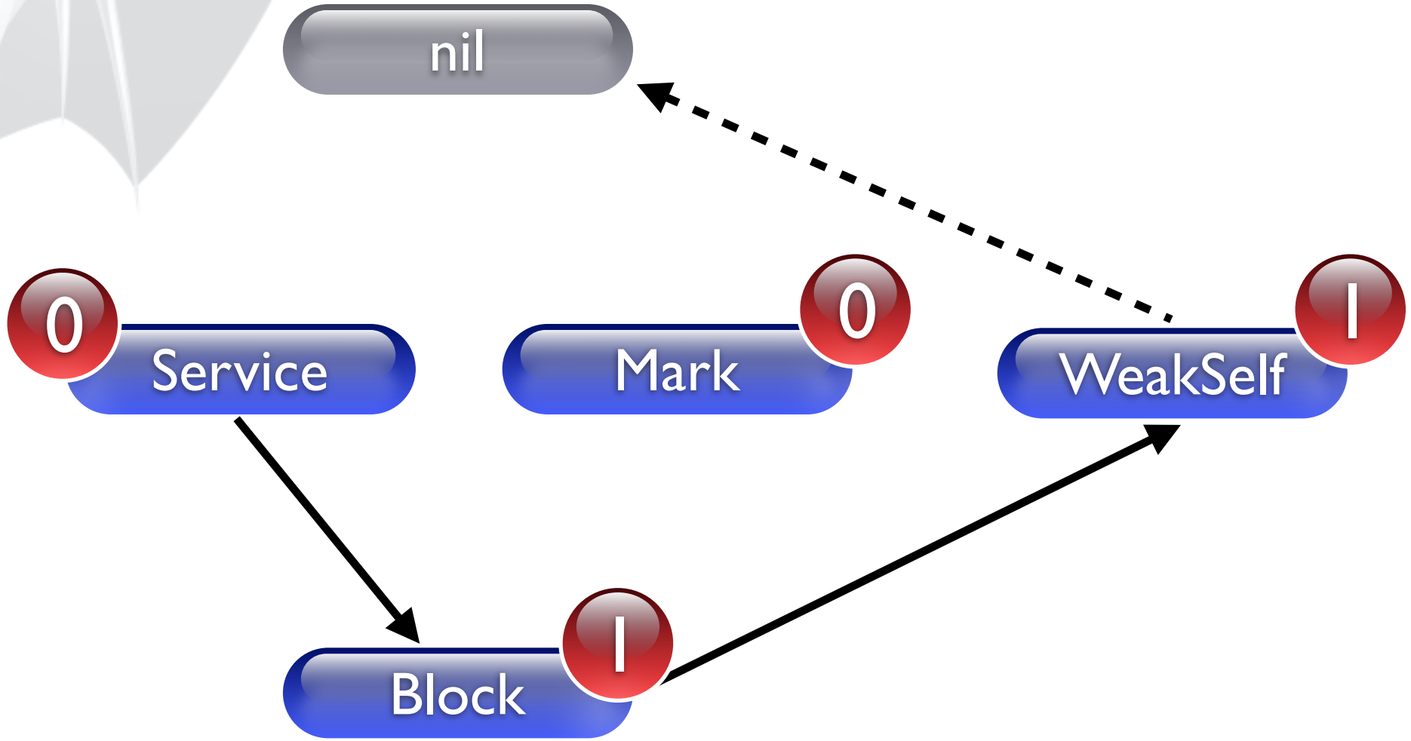
Blocks



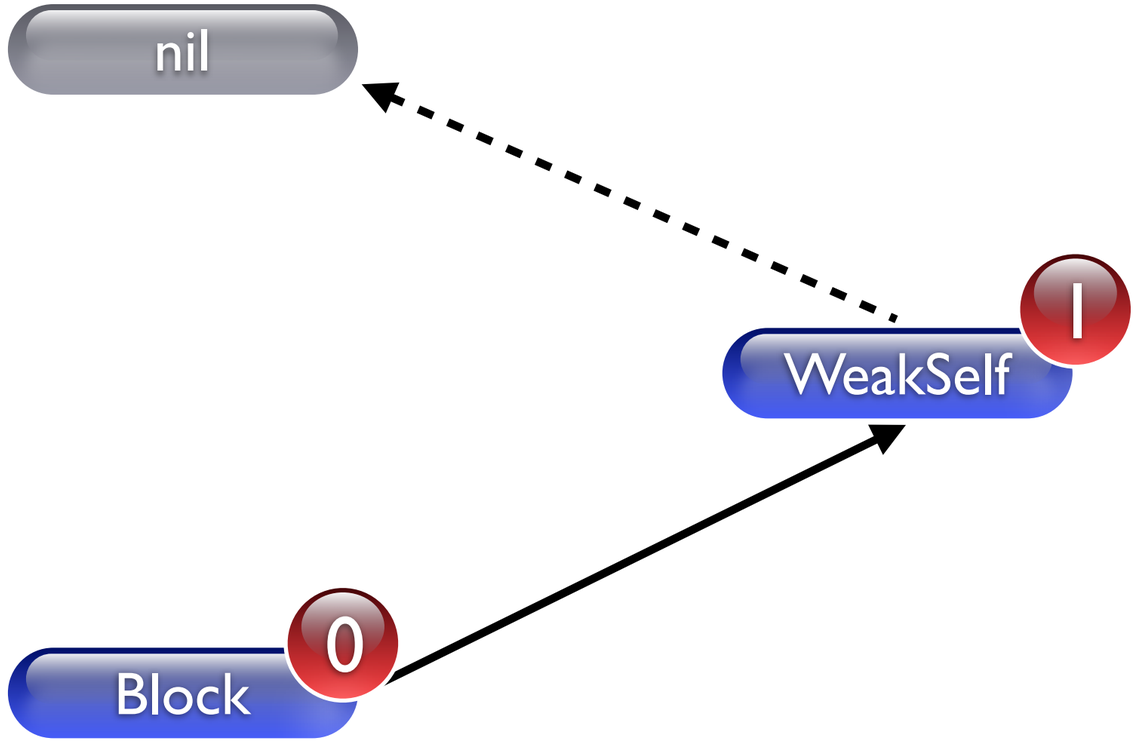
Blocks



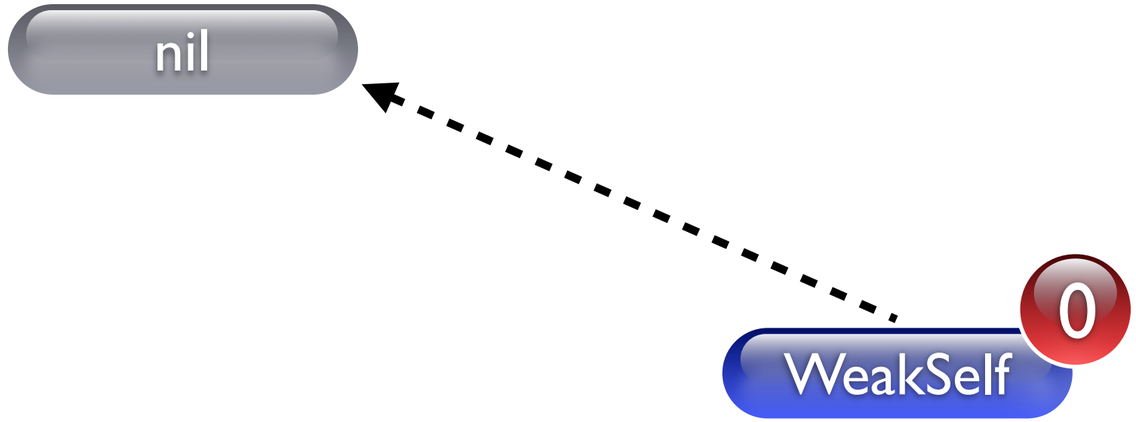
Blocks



Blocks



Blocks



Blocks

nil

Singletons

- Umstellen auf „shared Instance“:

```
+ (MyClass) sharedInstance {
  static MyClass *sharedMyClassInstance;
  static dispatch_once_t predicate;

  dispatch_once(&predicate, ^{
    sharedMyClassInstance = [[MyClass alloc] init];
  });

  return sharedMyClassInstanc;
}
```

Exceptions

- Nicht verwenden!
  - `__strong` Variablen werden nicht freigegeben
  - `__weak / __weak __block` werden abgemeldet
- Stattdessen Out-Parameter (`NSError **`) verwenden
- Wer Exceptions braucht: `-fobjc-arc-exception`
- Objective-C++: Exceptionhandling ist an.

Quellen

- *Quantifying the Performance of Garbage Collection vs. Explicit Memory Management*

  (http://www-cs.canisius.edu/~hertzm/gcmalloc-oopsla-2005.pdf)

Quellen

- *Automatic Reference Counting (ARC)*

  (http://clang.llvm.org/docs/AutomaticReferenceCounting.html)

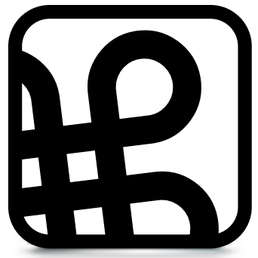
- *Programming with ARC Release Notes*

  (https://developer.apple.com/library/prerelease/ios/documentation/General/Conceptual/ARCProgrammingGuide/ARCProgrammingGuide.pdf)

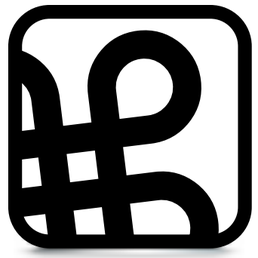
Quellen

- WWDC '11, Session 323, *Introducing Automatic Reference Counting*
- WWDC '11, Session 322, *Objective-C Advancements in Depth*
- WWDC '11, Session 316, *LLVM Technologies in Depth*

Fragen?



Danke!



Macoun’11