CS 696 Mobile Application Development
Fall Semester, 2010
Doc 12 Assignment 1 Comments Part 2
Oct 7, 2010

References

Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997

Problem 2

Interface

```
@interface NSNumber(NSNumber_Count)

-(NSNumber*) next;
+(int) callCount;

@end
```

```
#import "NSNumber-Count.h"

@implementation NSNumber (NSNumber_Count)

static int count = 0;

-(NSNumber*) next {
	count++;
	return [NSNumber numberWithInt:[self intValue]+1];
}

+(int) callCount {
	return count;
}

@end
```

The make it a function error

```
#import "NSNumber-Count.h"

@implementation NSNumber (NSNumber_Count)

static int count = 0;

-(NSNumber*) next: (NSNumber *) previous {
      count++;
      return [NSNumber numberWithInt:[previous intValue]+1];
}

+(int) callCount {
      return count;
}

@end
```

The make it a function error

```
NSNumber* test = [NSNumber numberWithInt:42];
NSNumber* nextNumber = [test next: test];
```

Problem 3

```
@protocol LinkedList

@property (readonly) NSNumber * count;

- (void) addFirst:(id)object;
- (void) addLast:(id)object;
- (void) removeFirst;
- (void) removeLast;

@end
```

Problem 4 & 5

Memory Policy

Node

- Owns data
- Retains/releases data

Linked list

- Owns nodes
- Retains/releases node

Node.h

```
@interface Node : NSObject {
	id value;
	Node * next;
}

@property (nonatomic,retain) id value;
@property (nonatomic,assign) id next;

- (id) initWithObject: (id) object;

@end
```

Node.m

```
@interface Node()
@end

@implementation Node

@synthesize value;
@synthesize next;

- (void) dealloc {
        [value release];
        [super dealloc];
}

@end
```

```
- (id) initWithObject: (id) object {
        if (self = [super init]) {
                self.value = object;     //retains object
                self.next = nil;
        }
}
```

SingleLinkedList.h

```
#import <Foundation/Foundation.h>
#import "LinkedList.h"
#import "Node.h"

@interface SingleLinkedList : NSObject <LinkedList> {
	Node *head;
	Node *tail;
	int size;
}

- (NSString *) description;
- (void) enumerateObjectsUsingBlock:(void (^)(id, NSUInteger, BOOL *))block;
- (id) objectAtIndex: (NSInteger) index;
- (id) initWithArray: (NSArray *) anArray;
+ (id) listWithArray: (NSArray *) anArray;

@end
```

The point of Problem 3 was to use the protocol in problem 4

SingleLinkedList.m first part

```
#import "SingleLinkedList.h"

@interface SingleLinkedList()
- (void) enumerateNodesUsingBlock:(void (^)(Node *, NSUInteger, BOOL *))block;
- (NSString *) componentsJoinedByString:(NSString *)separator;
- (Node *) nodeAt:(int) index;
@end

@implementation SingleLinkedList

- (NSNumber *) count {
      return [NSNumber numberWithInt: size];
}
```


It is a pain to use NSNumber internally so just use int. Generate NSNumber in count from size

Extensions (Empty Category)

Non public methods

Compiler requires methods to be implemented

No warning when used

```
@interface SingleLinkedList()
- (void) enumerateNodesUsingBlock:(void (^)(Node *, NSUInteger, BOOL *))block;
- (NSString *) componentsJoinedByString:(NSString *)separator;
- (Node *) nodeAt:(int) index;
@end

@implementation SingleLinkedList
```

```
- (id) init {
        if (self = [super init]) {
                head = nil;
                tail = nil;
                size = 0;
        }
        return self;
}
```

```
- (void) addFirst:(id)object {
        size++;
        Node *newNode = [[Node alloc] initWithObject: object];
        newNode.next = head;
        head = newNode;
        if(tail == nil) tail = newNode;
}
```

```
- (void) addLast:(id)object {
        size++;
        Node *newNode = [[Node alloc] initWithObject: object];
        tail.next = newNode;
        tail = newNode;
        if(head == nil) head = newNode;
}
```

```
- (void) removeFirst {
    if(size>0){
        Node *removeNode = head;
        size--;
        if(tail==head){
            tail = nil;
            head = nil;
        }else{
            head = head.next;
        }
        [removeNode release];
    }
}
```

```
- (void) removeFirst {
    if(size<1)
        return;
    Node *removeNode = head;
    head = head.next;
    [removeNode release];
  size--;
    if(size = 0){
        tail = nil;
        head = nil;
    }
}
```

```
- (NSString *) description {
        return [self componentsJoinedByString:@", "];
}
```

```
- (NSString *) componentsJoinedByString:(NSString *) separator {
    NSMutableString *resultString = [[NSMutableString alloc] init];
    [self enumerateObjectsUsingBlock:^(id obj, NSUInteger index, BOOL *stop){
        if(index>0){
            [resultString appendFormat:@"%@%@",separator,obj];
        } else {
            [resultString appendString:obj];
        }
    }];
    return [resultString autorelease];
}
```

```
- (id) objectAtIndex: (NSInteger) index {
        if(index<0 || index > size)
                @throw [NSException exceptionWithName: @"NSRangeException"
                                                reason: @"Out of range" userInfo: nil];
        return [self nodeAt:index].value;
}
```

```
- (id) initWithArray: (NSArray *) anArray {
	if (self = [self init]) {
		[anArray enumerateObjectsUsingBlock:^(id object, NSUInteger index, BOOL *stop)
{
			[self addLast:object];
		}];
	}
	return self;
}
```

```
+ (id) listWithArray: (NSArray *) anArray {
      return [[[SingleLinkedList alloc] initWithArray: anArray] autorelease];
}
```

What most people tried to do

```
- (void) enumerateObjectsUsingBlock:(void (^)(id, NSUInteger, BOOL *))block {
      int index = 0;
      BOOL stop = NO;
      Node *current = head;
      while (!stop && index<size) {
            block(current.value,index,&stop);
            current = current.next;
            index++;
      }
}
```

Shorter version

```
- (void) enumerateObjectsUsingBlock:(void (^)(id, NSUInteger, BOOL *))block {
      [self enumerateNodesUsingBlock:^(Node * current,NSUInteger index, BOOL * stop) {
          block(current.value,index,stop);
      }];
}
```

Work done here

```
- (void) enumerateNodesUsingBlock:(void (^)(Node *, NSUInteger, BOOL *))block {
    int index = 0;
    BOOL stop = NO;
    Node *current = head;
    while (!stop && index<size) {
        block(current,index,&stop);
        current = current.next;
        index++;
    }
}
```

No need for another loop

```
- (void) dealloc {
      [self enumerateNodesUsingBlock:^(Node * currentNode, NSUInteger index, BOOL *stop){
            [currentNode release];
      }];
      size = 0;
      [super dealloc];
}
```

No loop here either

```
- (Node *) nodeAt:(int) index {
	__block Node * nodeAtIndex;
	[self enumerateNodesUsingBlock:^(Node * current, NSUInteger loopIndex, BOOL *stop)
{
		if (index == loopIndex) {
			nodeAtIndex = current;
			*stop = YES;
		}
	}];
	return nodeAtIndex;
}
```

Kent Beck's Properties of Good Code Stype

Once and only once

Lots of little pieces

Replacing objects

Moving Objects

Rates of change

Once and Only Once

"In a program written with good style, everything is said once and only once"

If have

- several methods with same logic
- several objects with same methods

then rule is not satisfied

Lots of little pieces

"Good code invariably has small methods and small objects"

Small pieces allow you to satisfy "once and only once"

Primitive methods

Subset of methods of a class

All other method of class are implemented using primitive methods

Collection classes often have small set of primitive methods

- New type of collection only needs to implement primitive methods

enumerateNodesUsingBlock:

Only method that traverses nodes - once and only once

primitive method for SingleLinkedList class