

# Konzepte objektorientierter Programmierung

**Prof. Dr. Peter Müller**

**Werner Dietl**

Software Component Technology

Exercises 13: Specification

Wintersemester 03/04

**ETH**

Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Exercise 1: Singly-linked List

```
class List {  
    int i;  
    List next;  
  
    // invariant next != null => i == next.i  
  
    // i in der ganzen Liste erhöhen  
    void inc() {  
        ...  
    }  
}
```

## Specification of Method inc

```
// ensures  $\forall n \text{ reachable}(\mathbf{this}, n) :$   
//            $\text{old}(n.i) + 1 == n.i$   
void inc() { ... }
```

- The method reachable returns true, iff the second node can be reached from the first node
- After executing the method the old value of the element incremented by one equals the new value of the element

## Proof-Obligations for inc

- Proof postcondition fulfilled:  
 $\{ \mathbf{P} \wedge \forall S: INV_S \} \text{inc}() \{ \mathbf{Q} \}$
- Proof invariant still fulfilled:  
 $\{ \mathbf{P} \wedge \forall S: INV_S \} \text{inc}() \{ INV_{List} \}$
- The method inc has true as precondition
- The postcondition is shown on the last slide
- The invariant for the class was given

## Proof-Obligations for inc

$\{ \forall S: INV_S \}$

inc()

$\{ \forall n \text{ reachable}(\text{this}, n): \text{old}(n.i) + 1 == n.i \}$

$\{ \forall S: INV_S \}$

inc()

$\{ \forall x: \text{allocated}(x) \ \&\& \ \text{type}(x) <: \text{List}$

$\Rightarrow ( x.\text{next} \neq \text{null} \Rightarrow x.i == x.\text{next}.i ) \}$

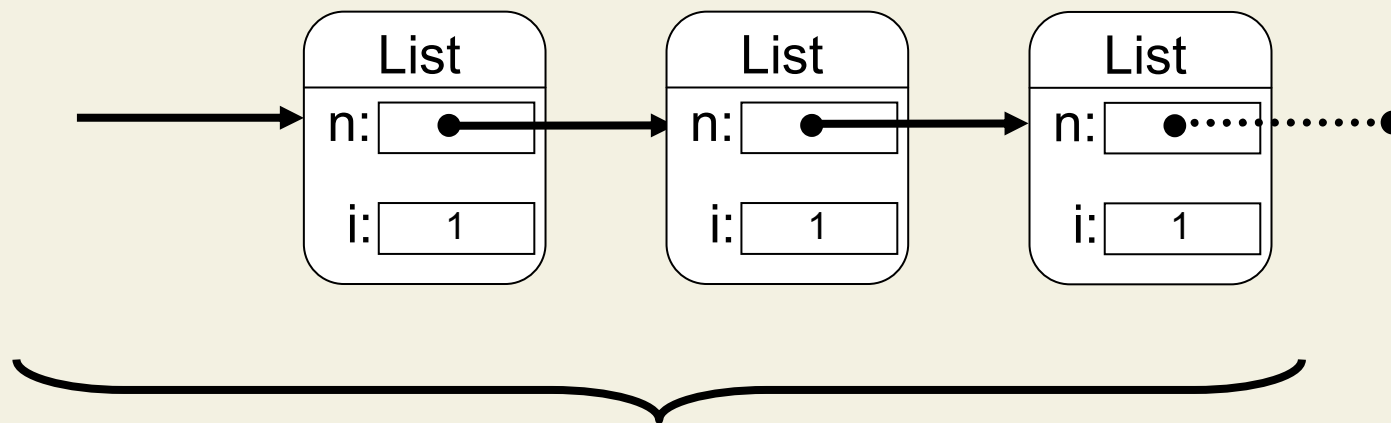
## Recursive Implementation of inc

```
void inc() {  
    ++i;  
    if( next != null ) next.inc();  
}
```

or

```
void inc() {  
    if( next != null ) next.inc();  
    ++i;  
}
```

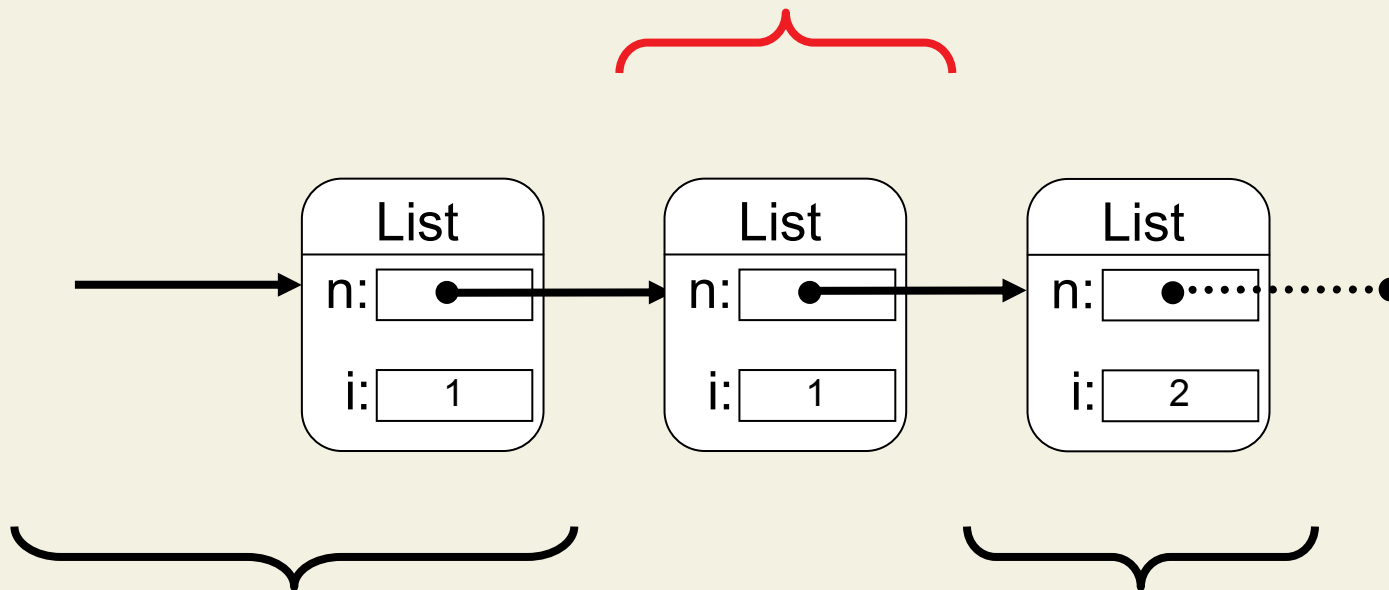
# Are the Proof-Obligations fulfilled?



**Invariant holds**

# Are the Proof-Obligations fulfilled?

**Invariant broken**

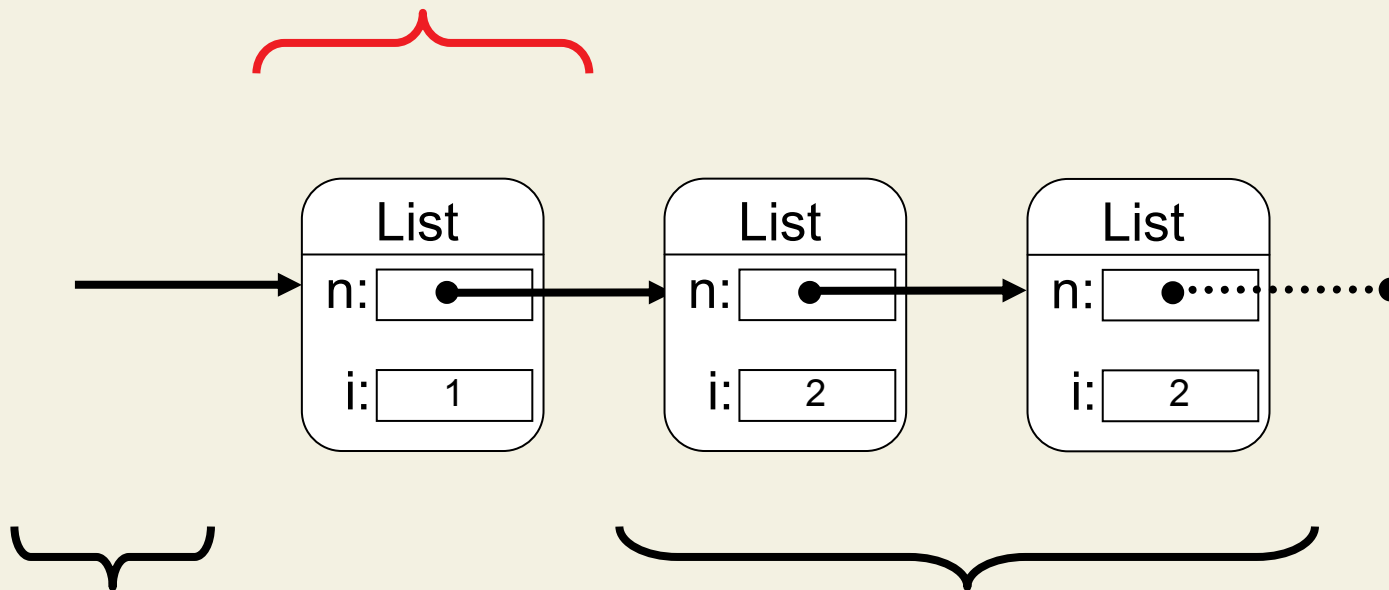


**Invariant holds**



# Are the Proof-Obligations fulfilled?

**Invariant broken**



**Invariant holds**

## Alternative Implementation 1

- Iterate over the list:

```
void inc() {  
    List n = this;  
  
    while( n != null ) {  
        ++n.i;  
        n = n.next;  
    }  
}
```

## Alternative Implementation 2

- Declare a private helper method `inc2` and allow it to break the invariant

```
public void inc() {  
    inc2();  
}
```

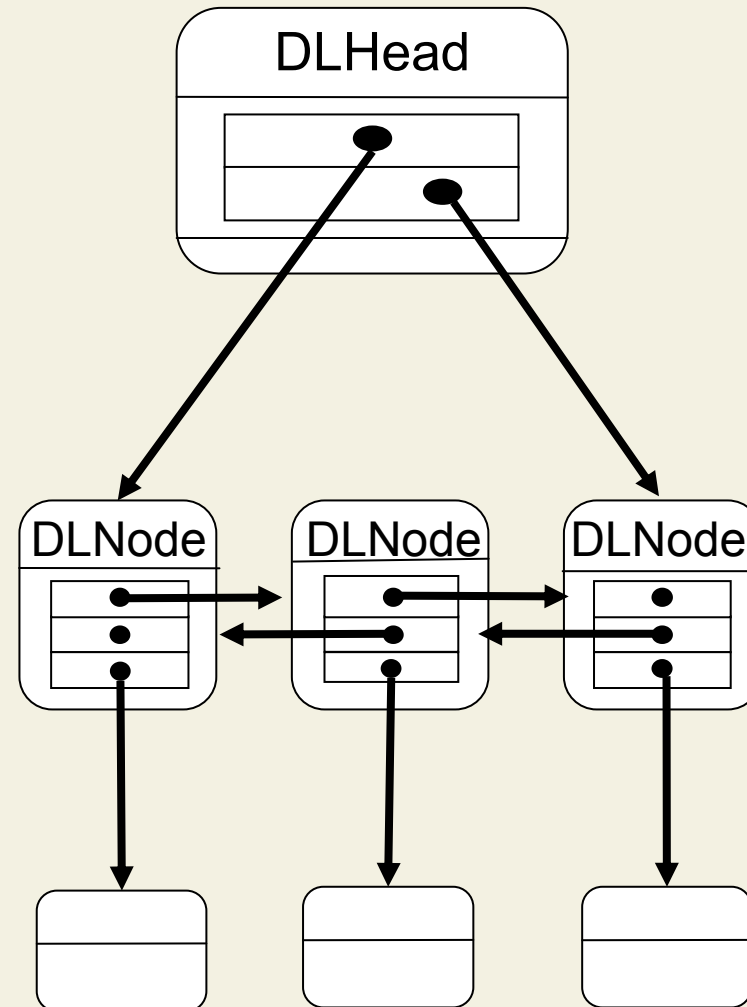
```
// helper
```

```
private void inc2() {  
    if( next != null ) next.inc2();  
    ++i;  
}
```

## Exercise 2: Doubly-linked List

```
public class DLHead {  
    private class DLNode {  
        DLNode prev;  
        DLNode next;  
        Object elem;  
    }  
  
    private DLNode first;  
    private DLNode last;  
    ...  
}
```

# What we want:



## Invariant 1: I'm my successor's predecessor

```
private class DLNode {  
    // invariant this.next != null =>  
    //             this.next.prev == this  
    ...  
}
```

- But is this enough?
- → No, `prev` could link into another List

## Invariant 2: I'm my predecessor's successor

```
private class DLNode {  
    // invariant this.next != null =>  
    //                this.next.prev == this  
    // invariant this.prev != null =>  
    //                this.prev.next == this  
    ...  
}
```

- Is this enough now?
- → No, we could have a circular list

## Invariant 3: first / last come / go nowhere

```
public class DLHead {  
    private DLNode first;  
    private DLNode last;  
  
    // invariant first != null =>  
    //             first.prev == null  
    // invariant last != null =>  
    //             last.next == null  
    ...  
}
```

- → No, we could have two independent lists



## Invariant 4: first and last use same list 1

```
public class DLHead {  
    private DLNode first;  
    private DLNode last;  
  
    // invariant first != null =>  
    //             first.next* == last  
    // or:  
    // invariant last != null =>  
    //             last.prev* == first  
    ...  
}
```

## Invariant 4: first and last use same list 2

```
public class DLHead {  
    private DLNode first;  
    private DLNode last;  
  
    // invariant reach( first, last )  
  
    boolean reach(DLNode from, DLNode to)  
    { ... }
```

## Using dummy-nodes

- Both the specification and implementation get simpler by always having a dummy first and last element
- These elements are always there and do not contain any data
- Example:

```
// invariant first.prev == null  
// invariant last.next == null
```

## Marrying a Man and a Woman

```
public class Man {  
    private Woman wife;  
  
    // invariant wife != null =>  
    //      wife.getHusband() == this  
  
    public boolean isMarried() {  
        return wife != null;  
    }  
    ...  
}
```

## Marrying a Man and a Woman

```
public class Woman {  
    private Man husband;  
  
    // invariant husband != null =>  
    //      husband.getWife() == this  
  
    public boolean isMarried() {  
        return husband != null;  
    }  
    ...  
}
```

## First Version

```
// Get married to 'w'
// require w != void && !w.isMarried()
// ensure isMarried() && wife == w
public void marry1( Woman w ) {
    wife = w;
    w.marry1( this );
}
```

- Doesn't work: call w.marry1 violates precondition

## Second Version

```
// Get married to 'w'
// require w != void && !w.isMarried()
// ensure isMarried() && wife == w
public void marry2( Woman w ) {
    w.marry2( this );
    wife = w;
}
```

- Doesn't work: Infinite recursion

## Third Version

```
// Get married to 'w'
// require w != void && !w.isMarried()
// ensure isMarried() && wife == w
public void marry3( Woman w ) {
    w.setHusband( this );
    wife = w;
}

// helper
void setWife(Woman w) { wife = w; }
```