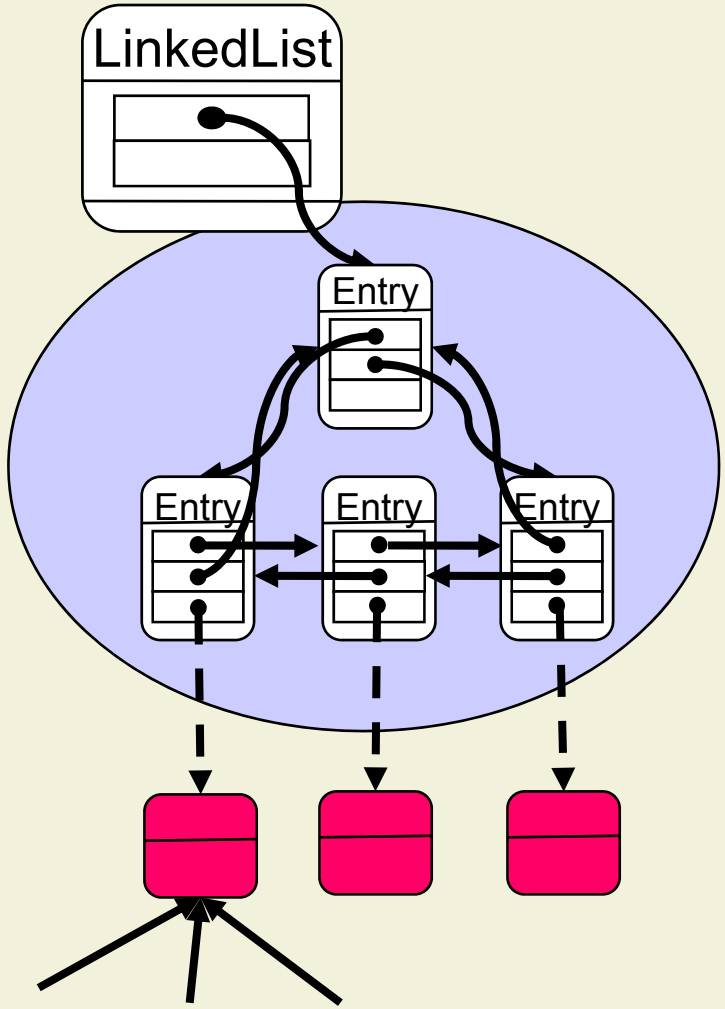
Concepts of Object-Oriented Programming

Prof. Dr. Peter Müller

Chair of Programming Methodology

Exercises 9: Ownership

Exercise 1 – LinkedList



Exercise 1 – Entries

```
class Entry {
  readonly Object element;
  peer Entry previous, next;

  Entry( readonly Object o,
     peer Entry p, peer Entry n ) {
   element = o;
   previous = p;
   next = n; }
}
```

Exercise 1 – LinkedList

```
public class LinkedList {
  private rep Entry header;
  private int size;

  public LinkedList() {
   header =
      new rep Entry(null, null, null);
   header.next = header;
   header.previous = header;
   size = 0;
  }
```

Exercise 1 – LinkedList

```
public void add( readonly Object o ) {
  rep Entry newE =
    new rep Entry(o,header,header.next);
  header.next.previous = newE;
  header.next = newE;
  ++size;
}
```

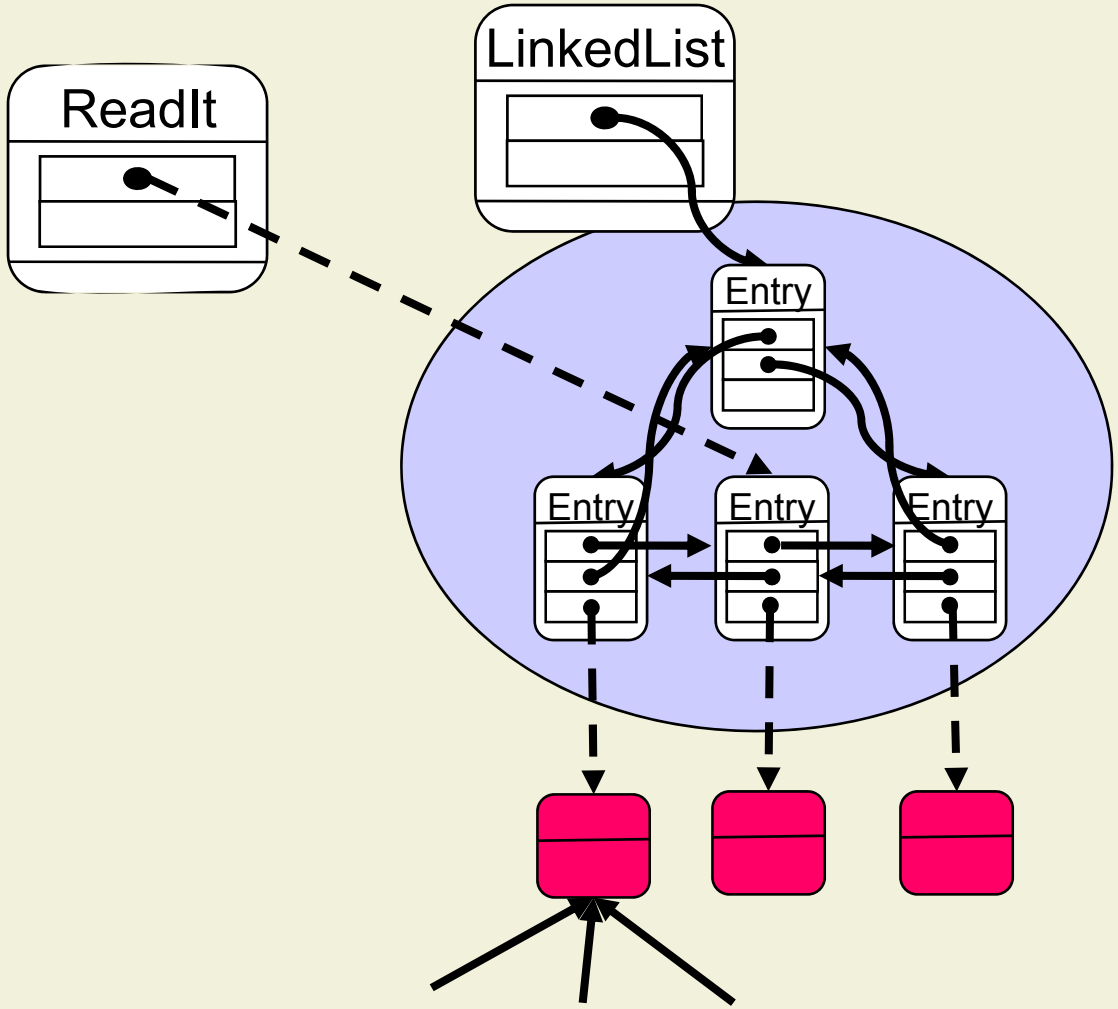
Exercise 1 – LinkedList

```
public pure readonly Object get(int idx){
  if( idx > size ) return null;

  readonly Entry e = header.next;
  for( int i=0; i<idx; ++i ) {
     e = e.next; }
  return e.element;
}
```

Exercise 1 – LinkedList

```
public pure peer ReadIterator
  getReadIterator() {
   return new peer ReadIterator(
                  header );
}
public pure peer DeleteIterator
  getDeleteIterator() {
   return new peer DeleteIterator(
                 this, header );
}
```

Exercise 1 – ReadIterator



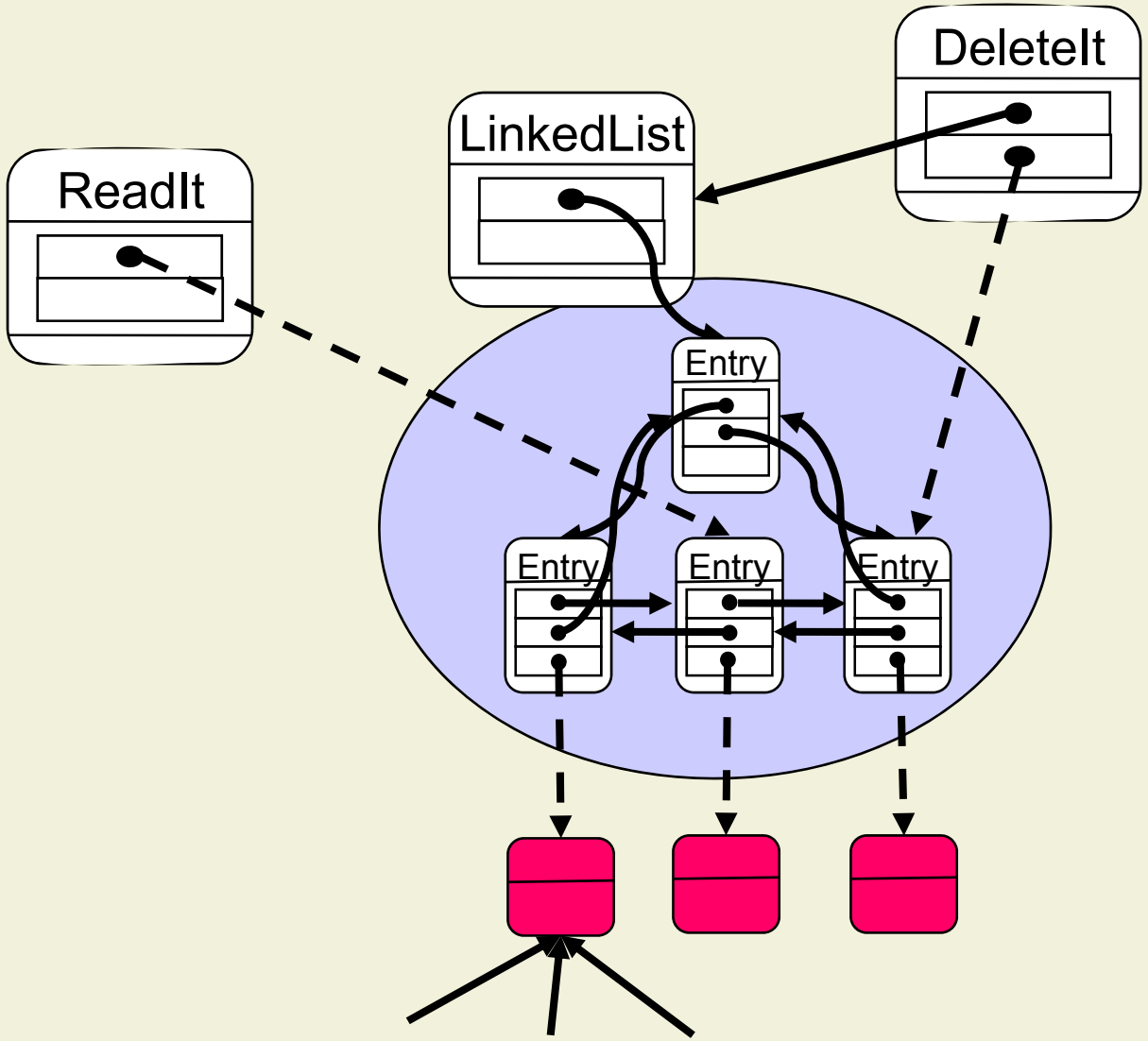
Exercise 1 – ReadIterator

```
public class ReadIterator {
  public ReadIterator(readonly Entry h)
  {
   // the header is a dummy that is
   // never null
   current = h;
   header = h;
  }
  public pure boolean hasNext() {
   return current.next != header; }
```

Exercise 1 – ReadIterator

```
  public void moveNext() {
      current = current.next;
  }
  public pure readonly Object element() {
      return current.element;
  }

  protected readonly Entry current;
  protected readonly Entry header;
}
```

Exercise 1 – DeleteIterator



Exercise 1 – LinkedList

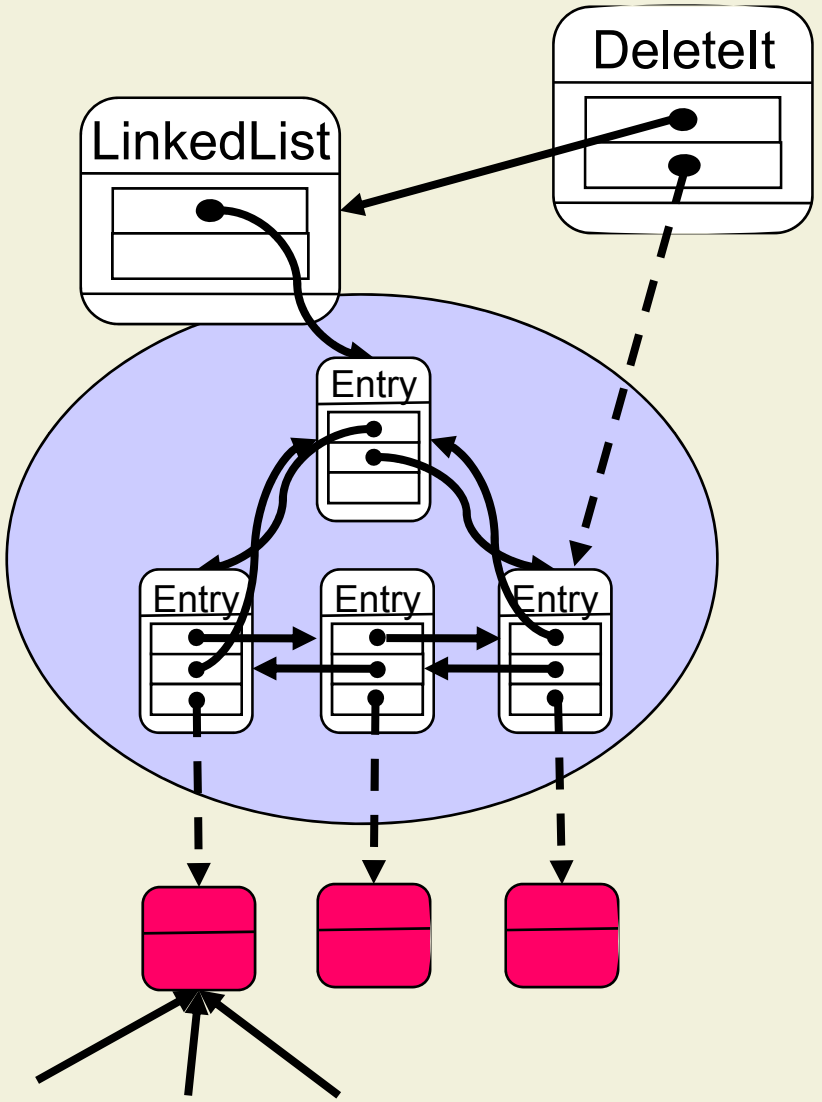
```
// precondition that the node belongs to us
/*@ requires e.owner == this; @*/
protected void delete( readonly Entry e ) {
  rep Entry re = (rep Entry) e;
  if( re.previous != null )
   re.previous.next = re.next;
  if( re.next != null )
   re.next.previous = re.previous;
  --size;
}
```

Exercise 1 – DeleteIterator

```
public class DeleteIterator
                  extends ReadIterator {
  public DeleteIterator( peer LinkedList l,
                         readonly Entry h ) {
     super( h ); list = l;
  }
  public void delete() {
     list.delete(current);
     current = current.next;
  }
  private peer LinkedList list;
}
```

Exercise 1 – LinkedList

- Entries are encapsulated
- Readonly access possible
- No external modifications
- Iterator has to delegate the modification to the LinkedList
- LinkedList is in control



Exercise 2: List with merge

```
class Entry {
    Object element;
    Entry previous, next;
    Entry(Object o, Entry p, Entry n ) {...}
}

class LinkedList {
    private Entry header;
    ...
}
```

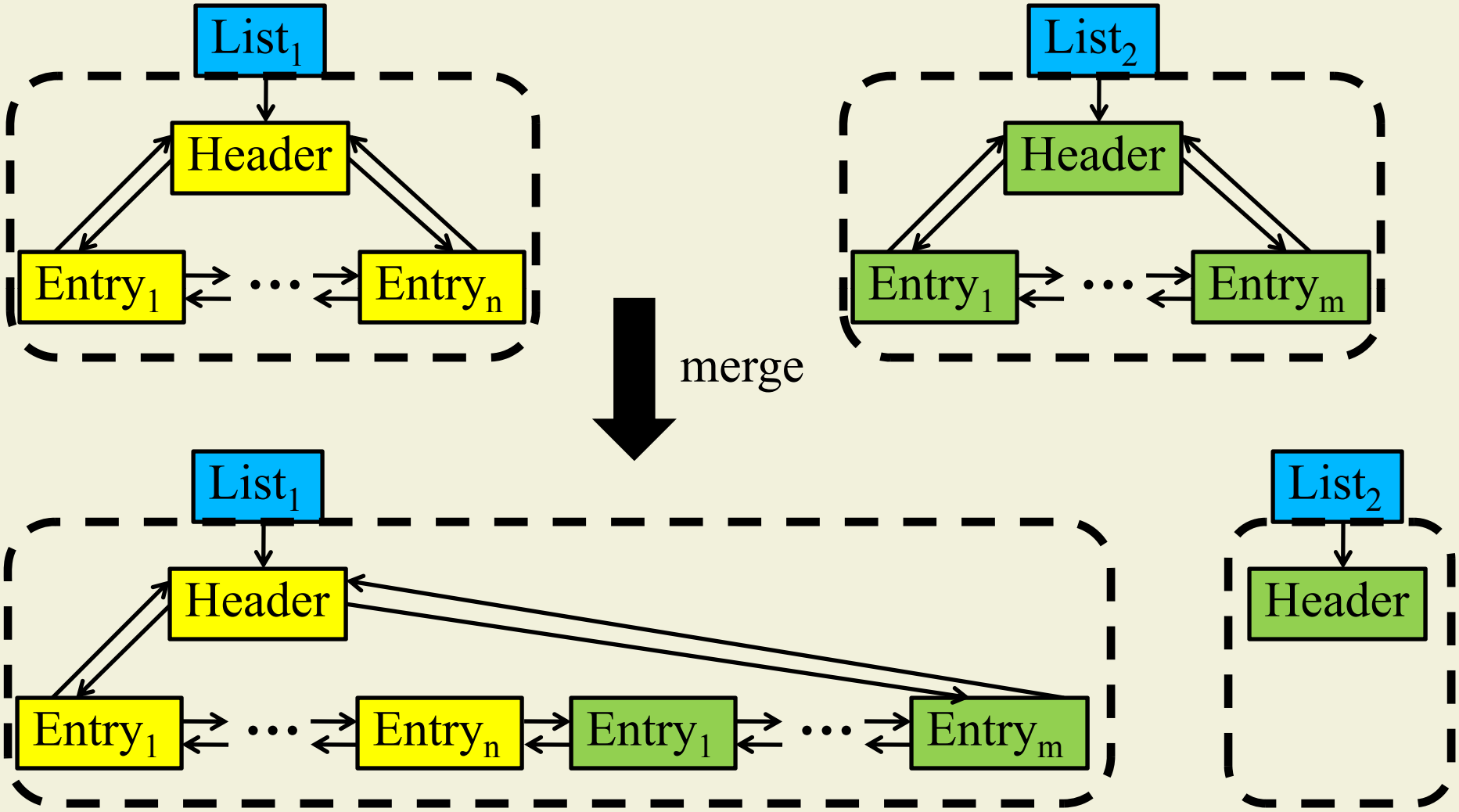
Exercise 2: List with merge

```
class Entry {
    readonly Object element;
    peer Entry previous, next;
    Entry(readonly Object o,
        peer Entry p, peer Entry n ) {...}
}

class LinkedList {
    rep private Entry header;
    ...
}
```

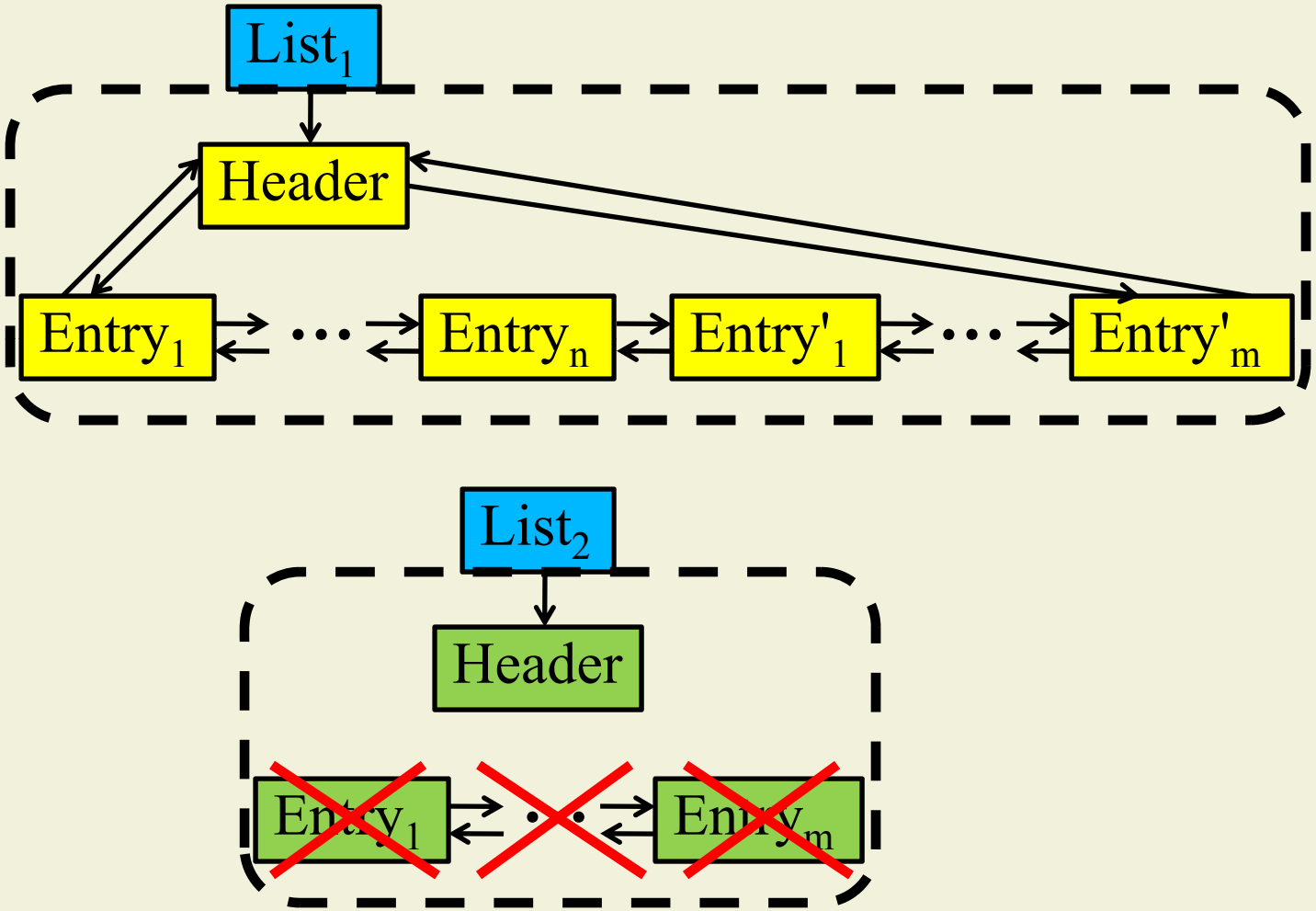
Exercise 2

```
public void merge(LinkedList other) {
    if (other.isEmpty()) {return;}

    Entry first = other.getHeader().next;
    Entry last = other.getHeader().previous;
    header.previous.next = first;
    last.next = header;
    first.previous= header.previous;
    header.previous = last;

    other.Init();
}
```

Exercise 2: merge



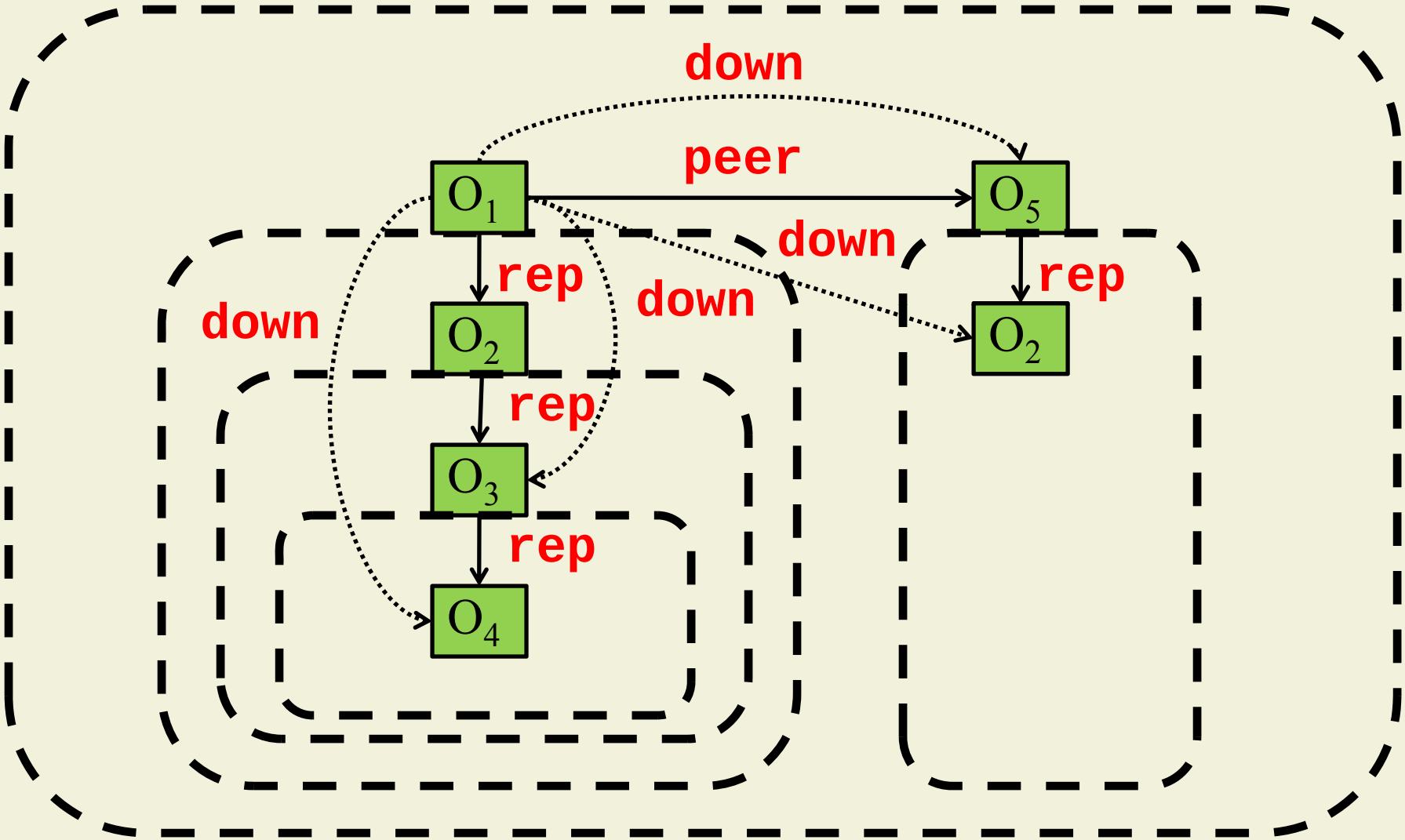
Exercise 2: modified merge



Exercise 2

```
public void merge(LinkedList other) {
    if (other.isEmpty()) {return;}

    readonly Entry entry =
        other.getHeader().next;
    while(entry != other.getHeader()){
        add(entry.element);
        entry = entry.next;
    }
    other.Init();
}
```

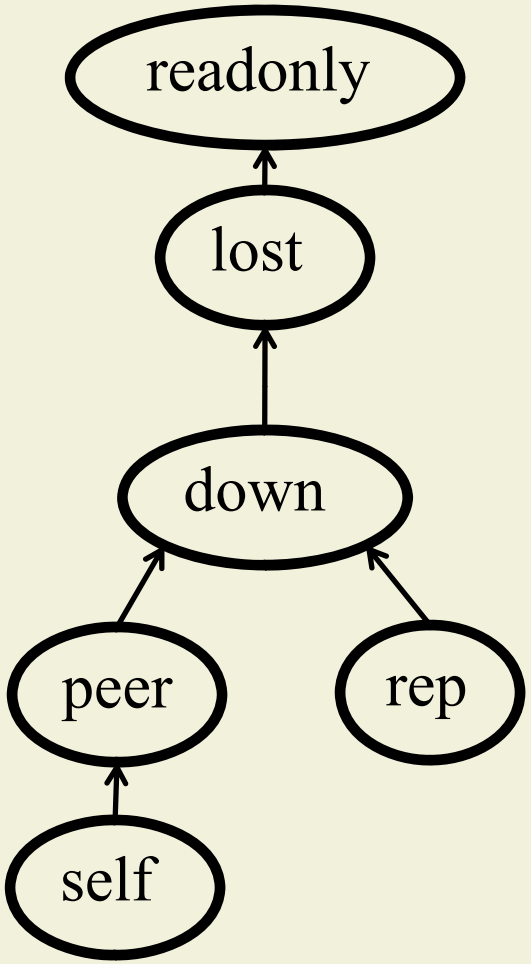
Exercise 3



Exercise 3.a

►	peer	rep	lost	ro	down
self	peer	rep	lost	ro	down
peer	peer	down	lost	ro	down
rep	rep	down	lost	ro	down
lost	lost	lost	lost	ro	lost
ro	lost	lost	lost	ro	lost
down	down	down	lost	ro	down

Exercise 3.b



Exercise 3.c

- The field read

`v = exp.f;`

is correctly typed if
- exp is correctly typed
- $\tau(exp) \blacktriangleright \tau(f) <: \tau(v)$

- The field write

`exp.f = v;`

is correctly typed if
- exp is correctly typed
- $\tau(v) <: \tau(exp) \blacktriangleright \tau(f)$
- **lost** not in $\tau(exp) \blacktriangleright \tau(f)$
- **down** not in $\tau(exp) \blacktriangleright \tau(f)$

- Analogous rules are used for method invocations

Questions?