AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture & Design of Embedded Real-Time Systems (TI-AREM)

Memory Patterns

B.D. Chapter 6. 259-300

Version: 9-3-2015

Agenda

Introduction – what is the problem

1. Static Allocation Pattern
2. Pool Allocation Pattern
3. Fixed Sized Buffer Pattern
4. Smart Pointer Pattern
5. Garbage Collection Pattern
6. Garbage Compactor Pattern

Introduction – what is the problem?

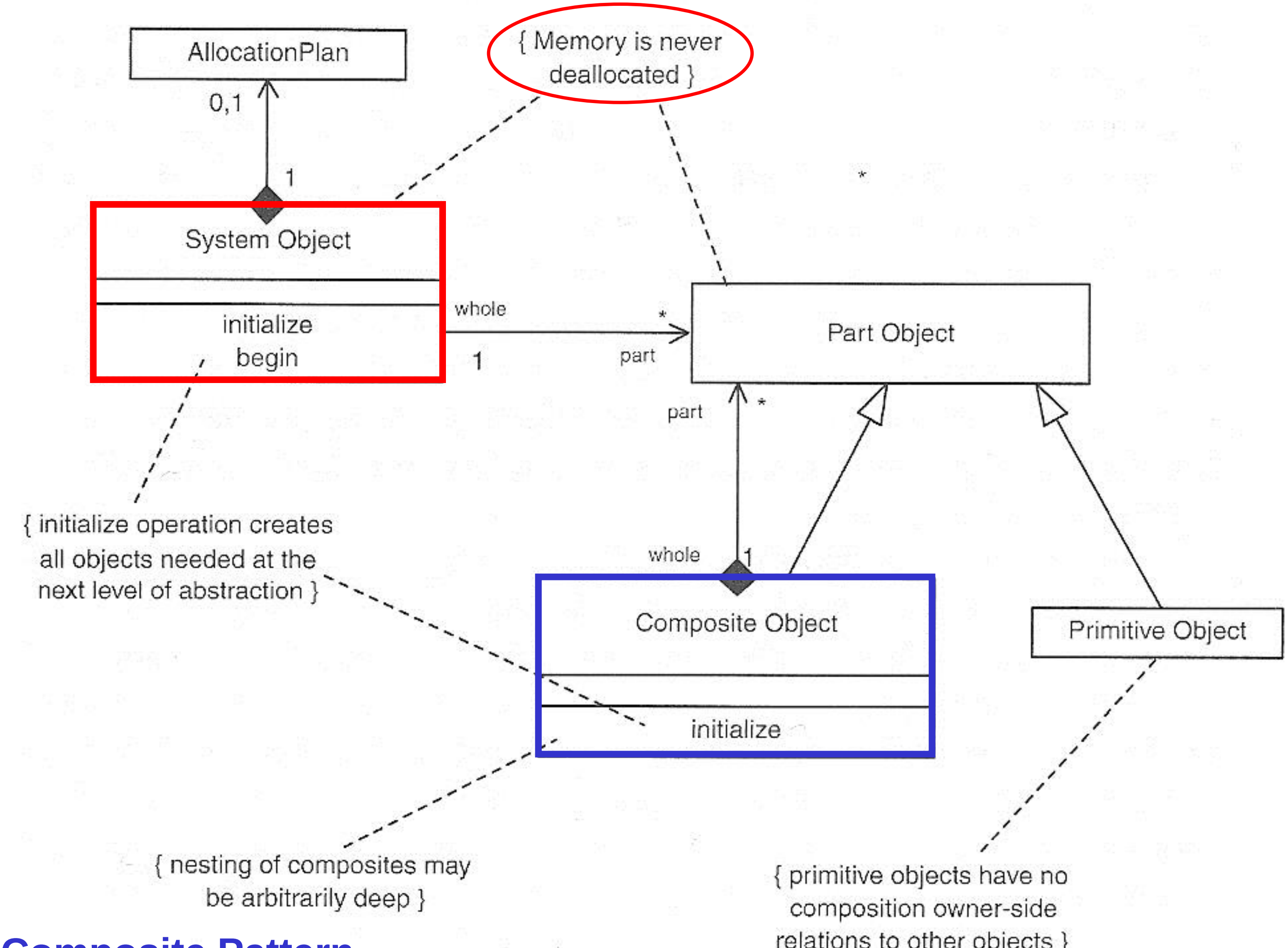
- Real-time systems must have **predictable behavior**
- Dynamic memory allocation with new uses the heap, which have unpredictable behavior
- A general solution: **avoid use of dynamic memory allocation** (i.e. new of objects on the heap) - at least in the real-time loops
- Use one of the following memory patterns

1. Static Allocation Pattern

The Static Allocation Patterns **disallows dynamic memory allocation during the application real-time loops**

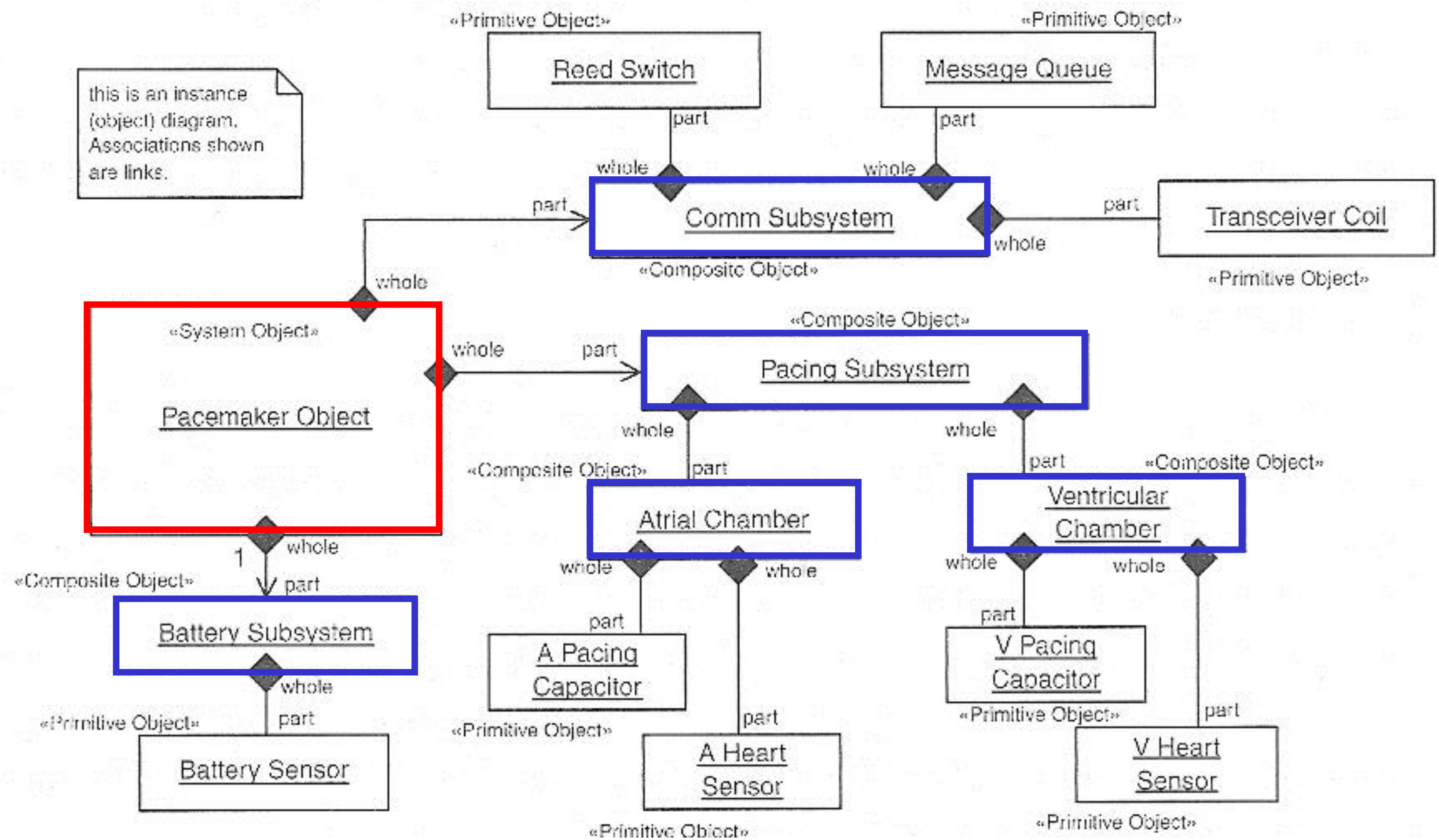
All objects are **allocated during system initialization**

Static Allocation Pattern Structure

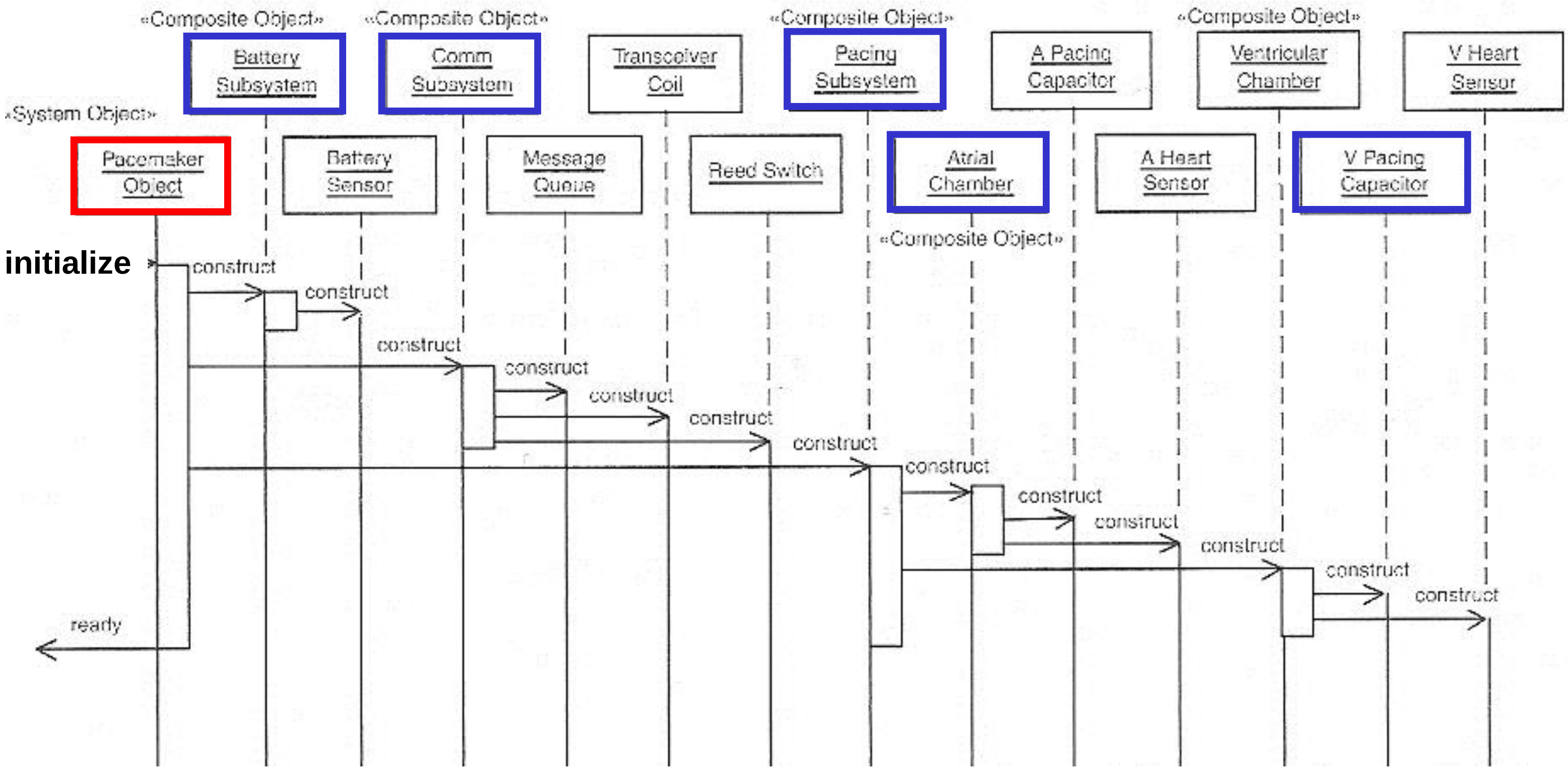


GoF Composite Pattern

Static Allocation Pattern Example (1)



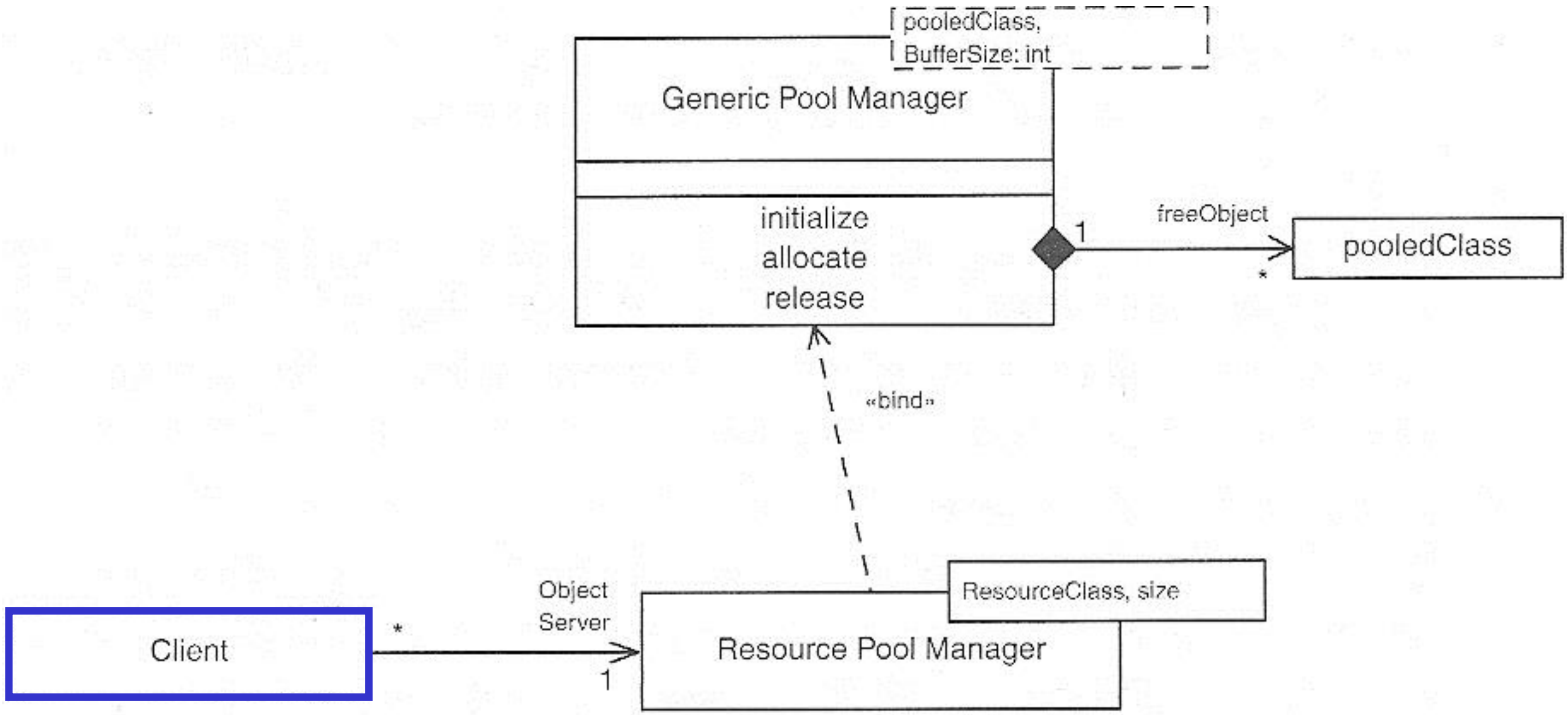
Static Allocation Pattern Example (2)



2. Pool Allocation Pattern

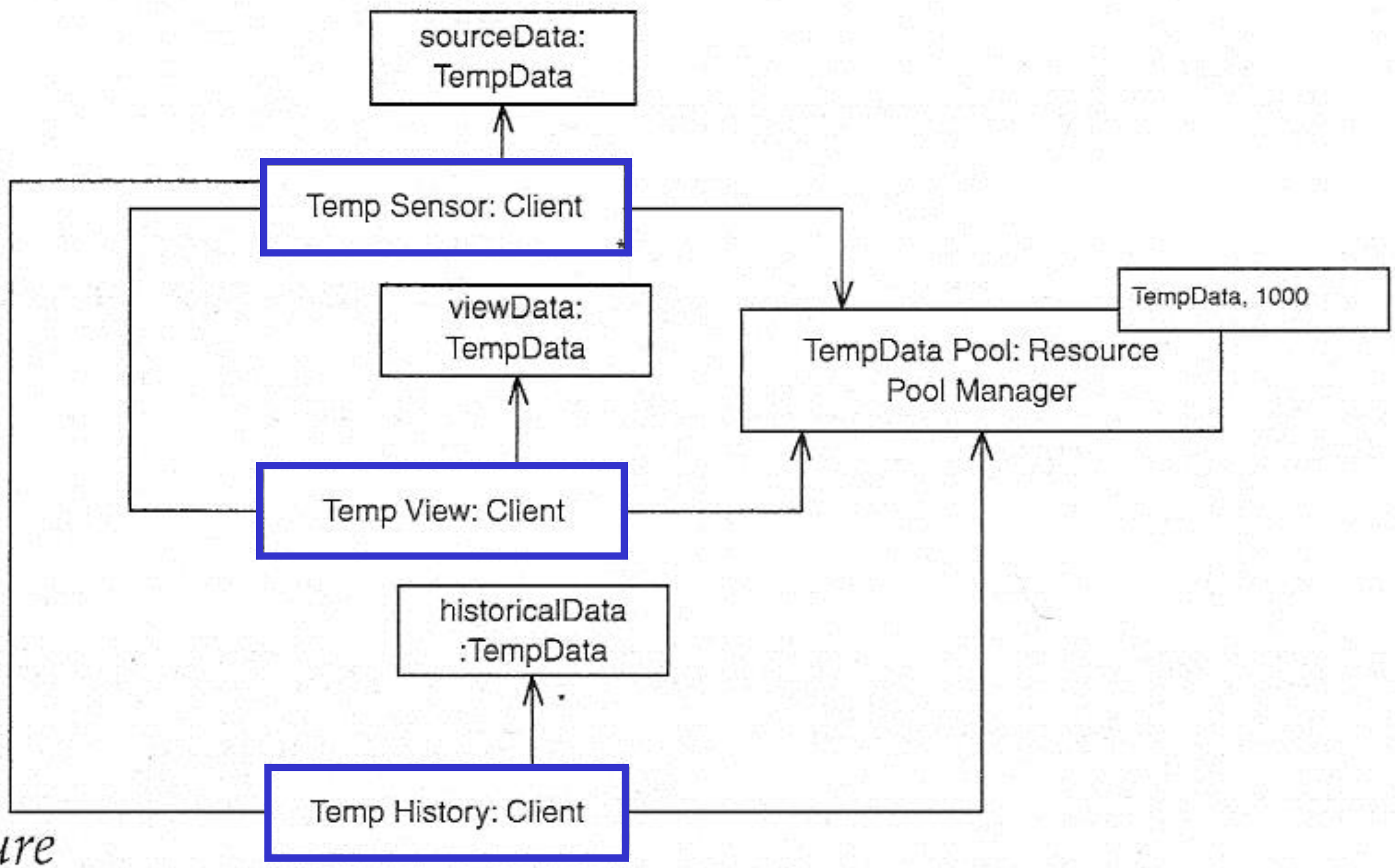
The Pool Allocation Patterns creates pools of objects, **created at startup**, available to clients upon request

Pool Allocation Pattern Structure



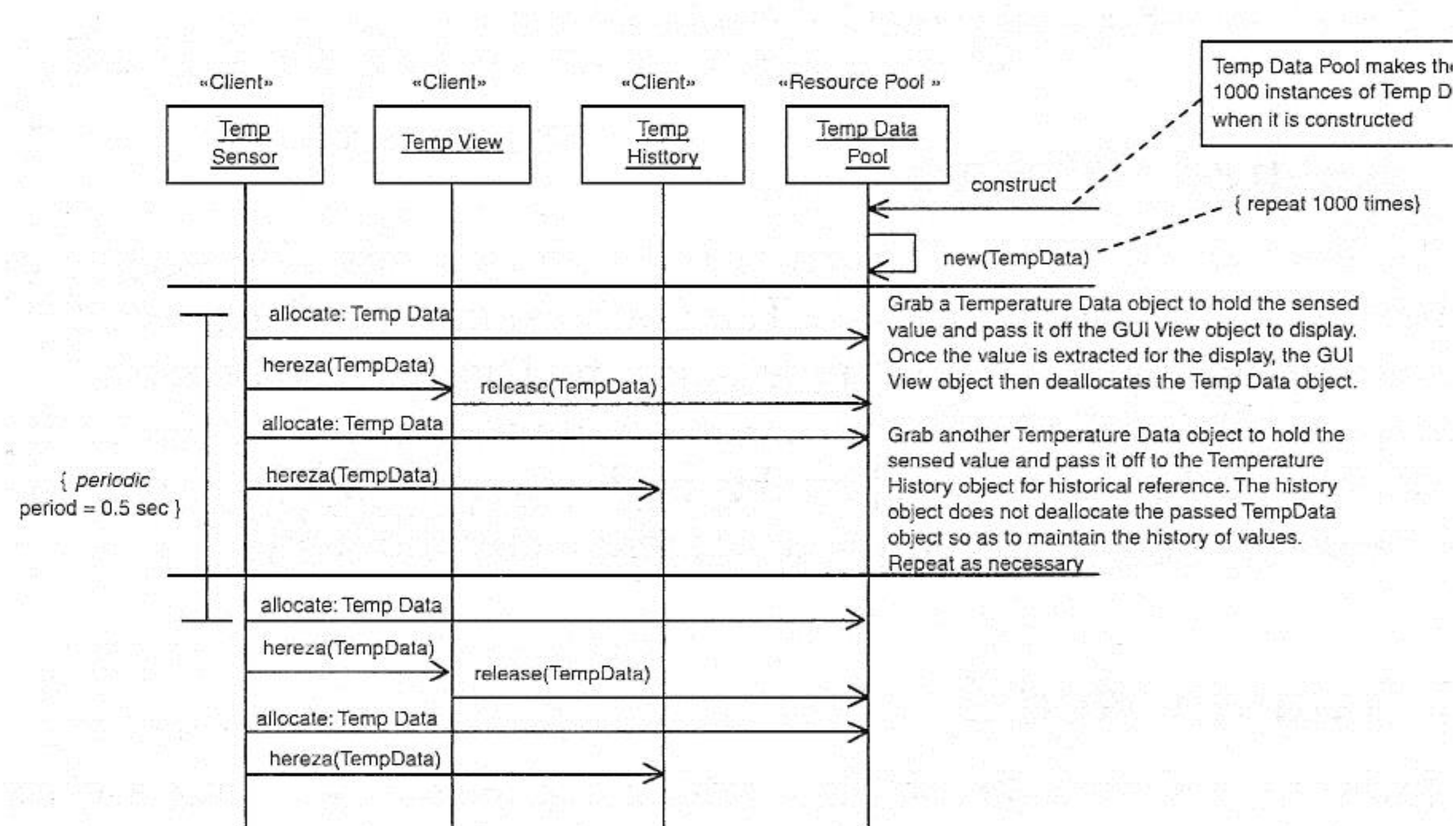
UML Template Class Notation

Pool Allocation Pattern Example (1)



a. Structure

Pool Allocation Pattern Example (2)



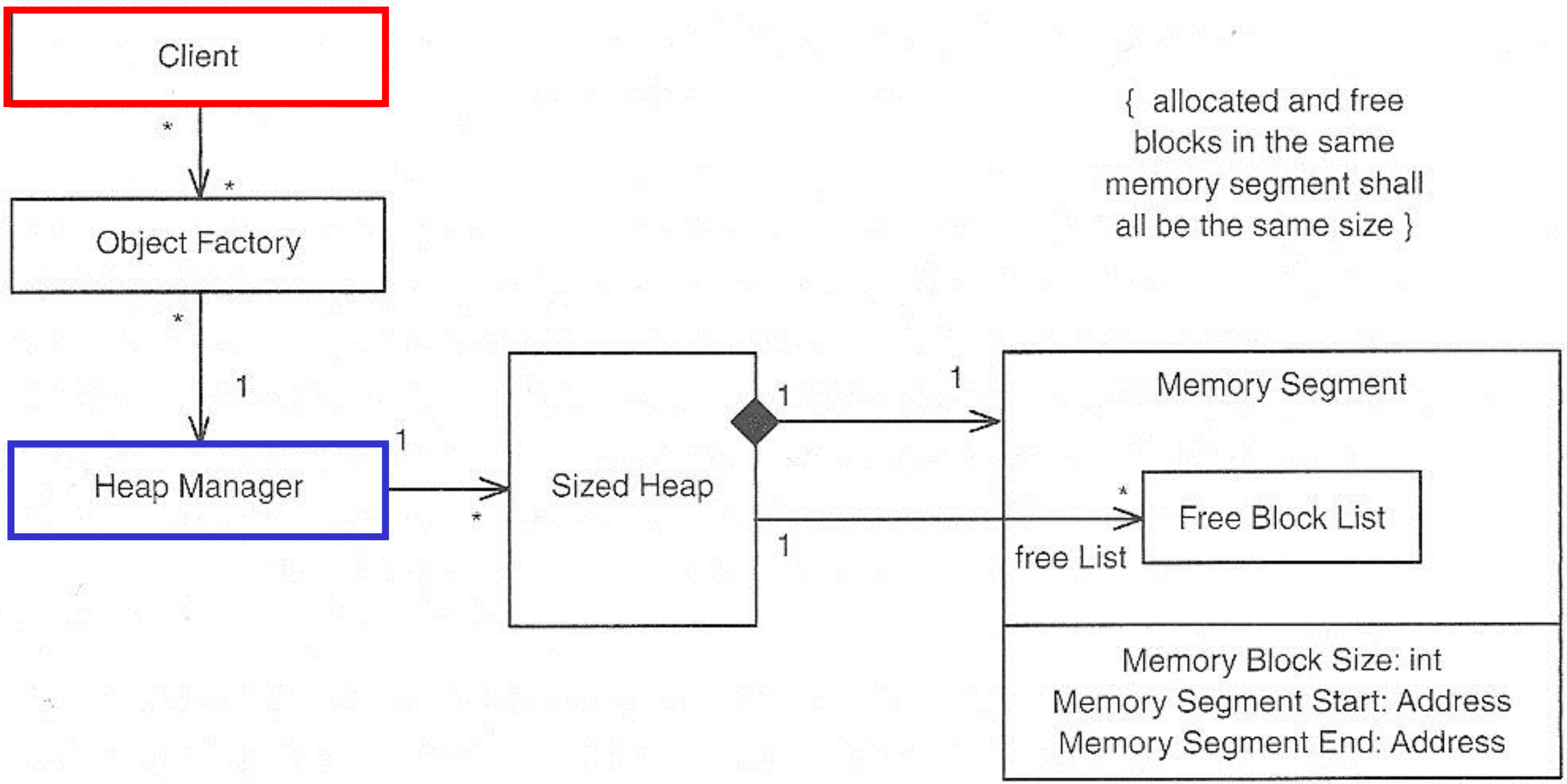
b. Scenario

3. Fixed Sized Buffer Pattern

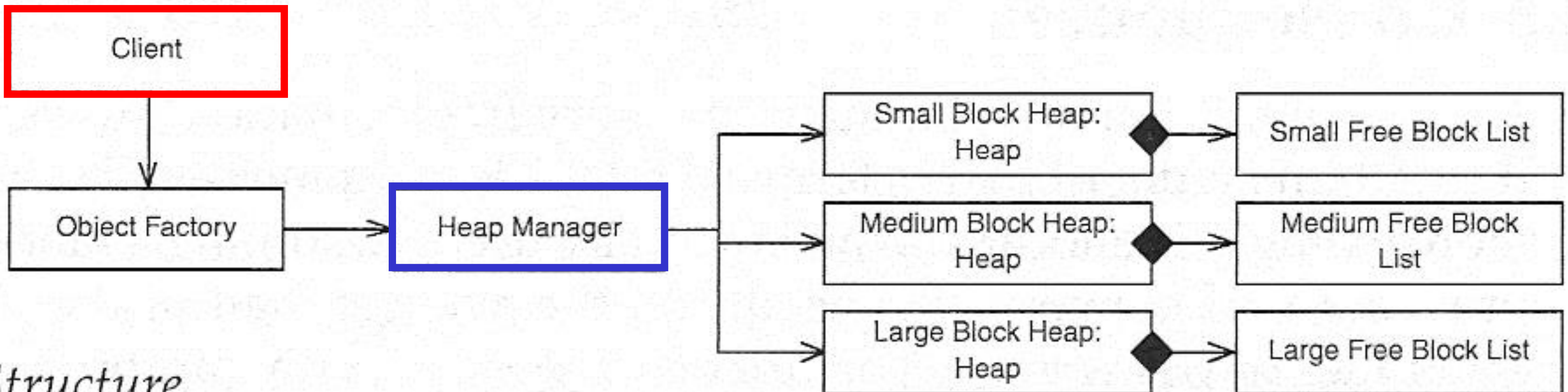
The basic concepts of the Fixed Sized Buffer Pattern is not to allow memory to be allocated in any random block size but **to limit the allocation to a set of specific block sizes**

It is a pattern supported directly by most Real-time Operating Systems

Fixed Sized Buffer Pattern Structure

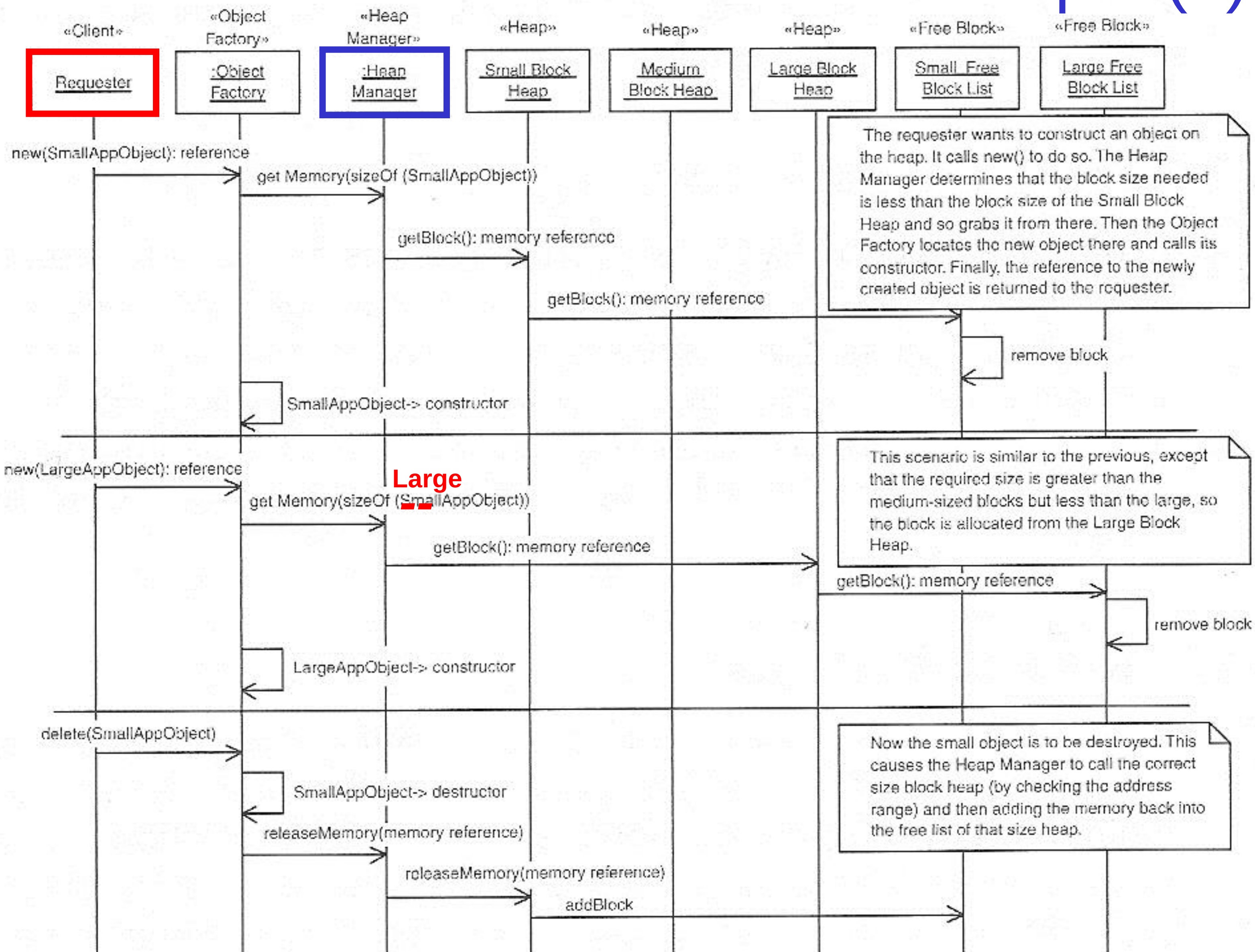


Fixed Sized Buffer Pattern Example (1)



a. Structure

Fixed Sized Buffer Pattern Example (2)



b. Scenario

4. Smart Pointer Pattern

The Smart Pointer Pattern **makes the pointer itself an object**
Solves a number of typical C++ problems with pointers

What are Smart Pointers

- Smart pointers are objects that look and feel like pointers, but are smarter
- To look and feel like pointers, smart pointers need to have the same interface that pointers do:
  - they need to support pointer operations like dereferencing (operator *) and indirection (operator ->)
- An object that looks and feels like something else is called a **proxy** object

A Smart Pointer Example

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr() {delete ptr;}
    T& operator*() {return *ptr;}
    T* operator->() {return ptr;}
    // ...
};
```

auto_ptr is a class in the C++ Standard Template Library

explicit Keyword in C++

The explicit keyword

C++ adopted a new **explicit** keyword to the language.

This keyword is a qualifier used when declaring constructors. When a constructor is declared as **explicit**, the compiler will never call that constructor implicitly as part of a type conversion.

This allows the compiler to perform stricter type checking and to prevent simple programming errors.

If your compiler does not support the **explicit** keyword, you should avoid it and do without the benefits that it provides.

Using a Smart Pointer

```
void foo()
{   // using a normal pointer
    MyClass* p(new MyClass);
    p->DoSomething();
   delete p;
}
```

```
void foo()
{  // using a smart pointer
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
}
```

Dangling Pointers

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
p->DoSomething(); // Watch out! p is now dangling!
p = NULL;         // p is no longer dangling
q->DoSomething(); // Ouch! q is still dangling!
```

auto_ptr Assignment Operator

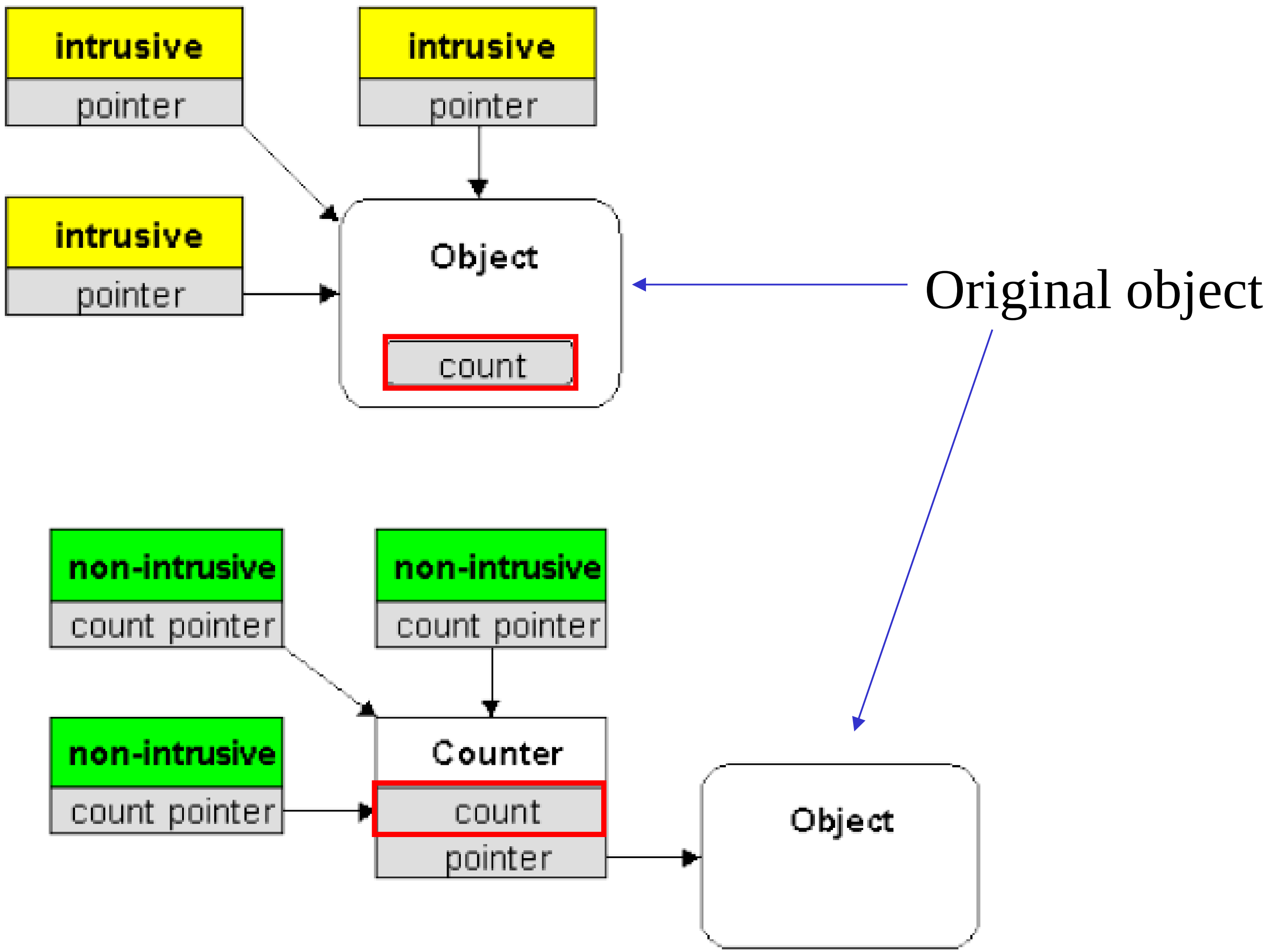
For auto_ptr, dangling is avoided by setting its pointer to NULL when it is copied

```
template <class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
    if (this != &rhs)
    {
        delete ptr;
        ptr = rhs.ptr;
        rhs.ptr = NULL;
    }
    return *this;
}
```

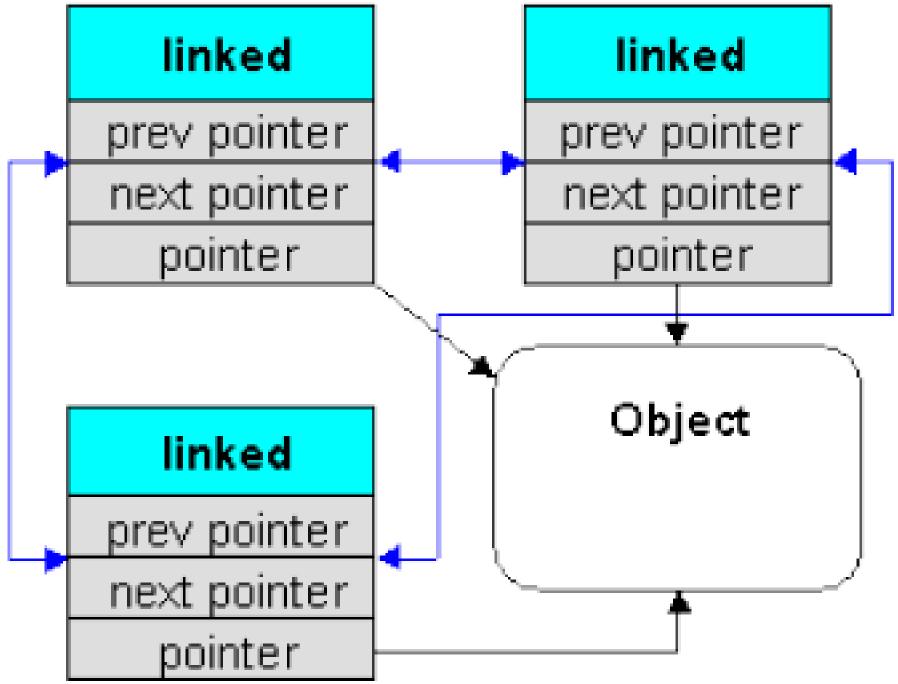
Assignment Strategies for q=p

- **Create a new copy:** of the object pointed by p, and have q point to this copy
- **Ownership transfer**: Let both p and q point to the same object, but transfer the responsibility for cleaning up ("ownership") from p to q
- **Reference counting**: Maintain a count of the smart pointers that point to the same object, and delete the object when this count becomes zero
- **Reference linking**: The same as reference counting, only instead of a count, maintain a circular doubly linked list of all smart pointers that point to the same object
- **Copy on write**: Use reference counting or linking as long as the pointed object is not modified. When it is about to be modified, copy it and modify the copy

Reference counting principles (1)



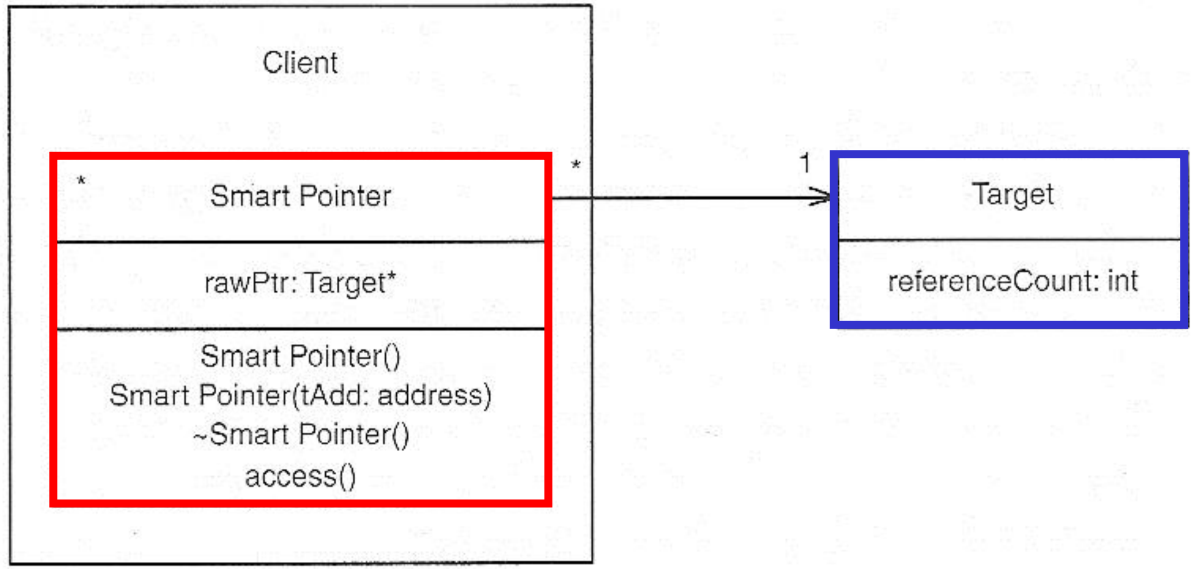
Reference counting principles (2)



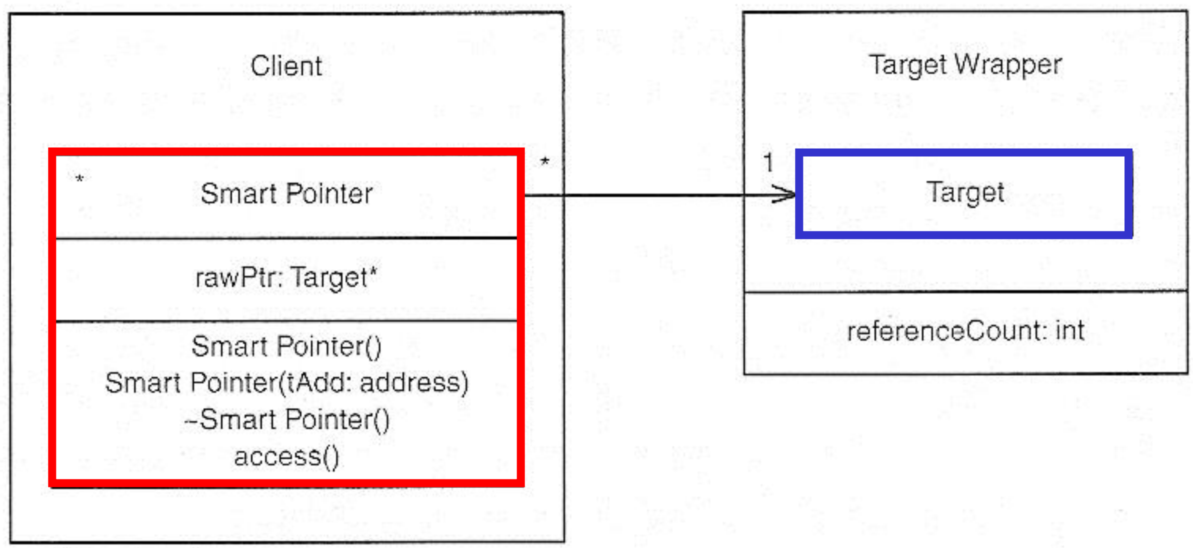
Reference linking does not require any changes to be made to the pointed objects, nor does it require any additional allocations.

A reference linked pointer takes a little more space than a reference counted pointer – just enough to store one or two more pointers.

Smart Pointer Pattern Structures

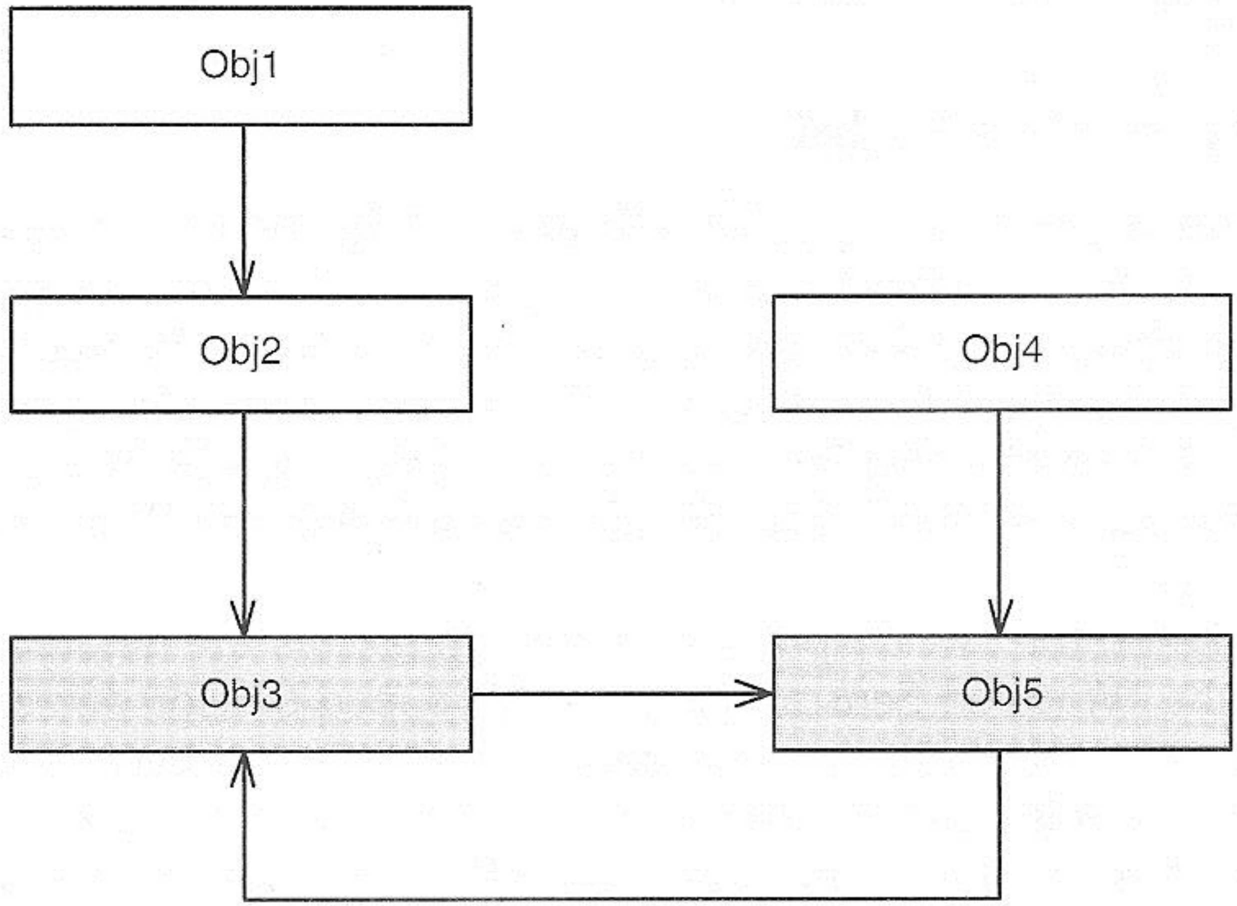


a. Basic



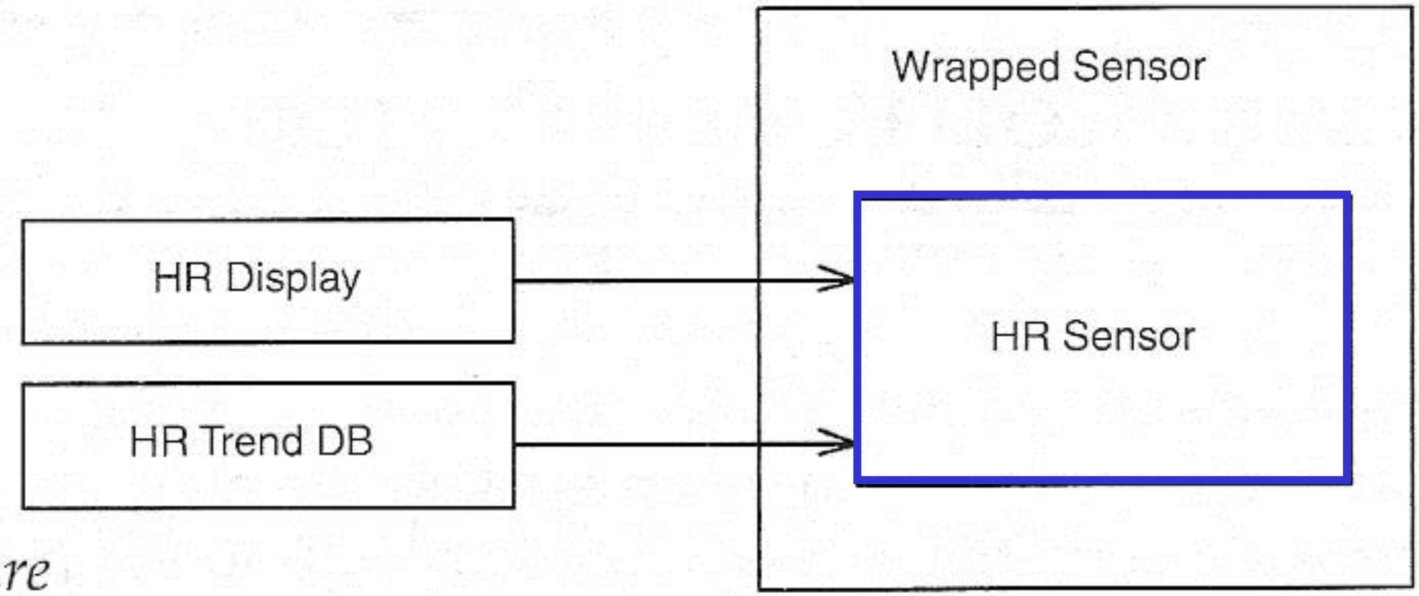
b. Wrapper Variant

Smart Pointer Cycles



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

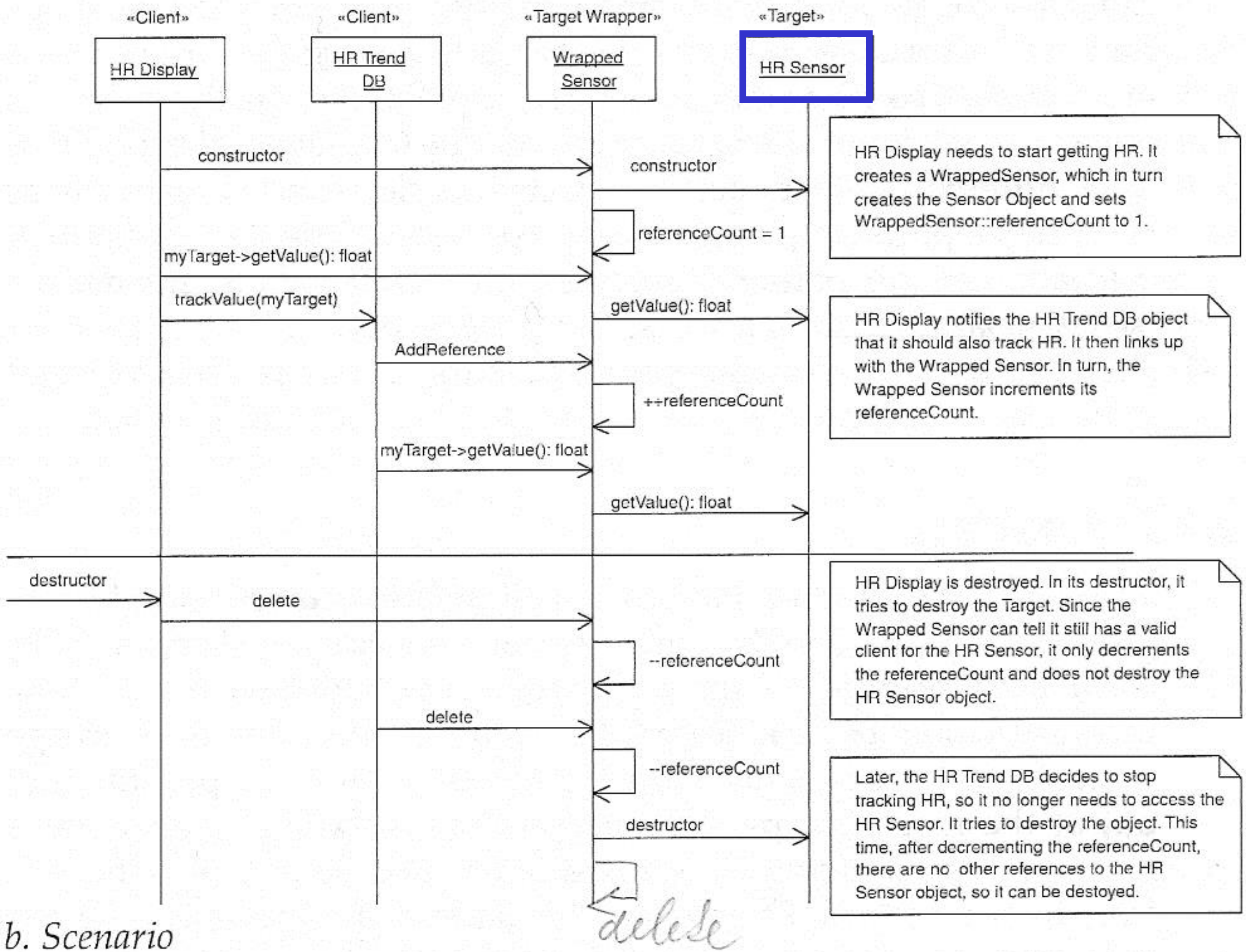
Smart Pointer Pattern Example (1)



a. Structure

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Smart Pointer Pattern Example (2)



b. Scenario

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Links

- The Boost C++ libraries include some smart pointers. http://www.boost.org/

The smart pointer library provides six smart pointer class templates:

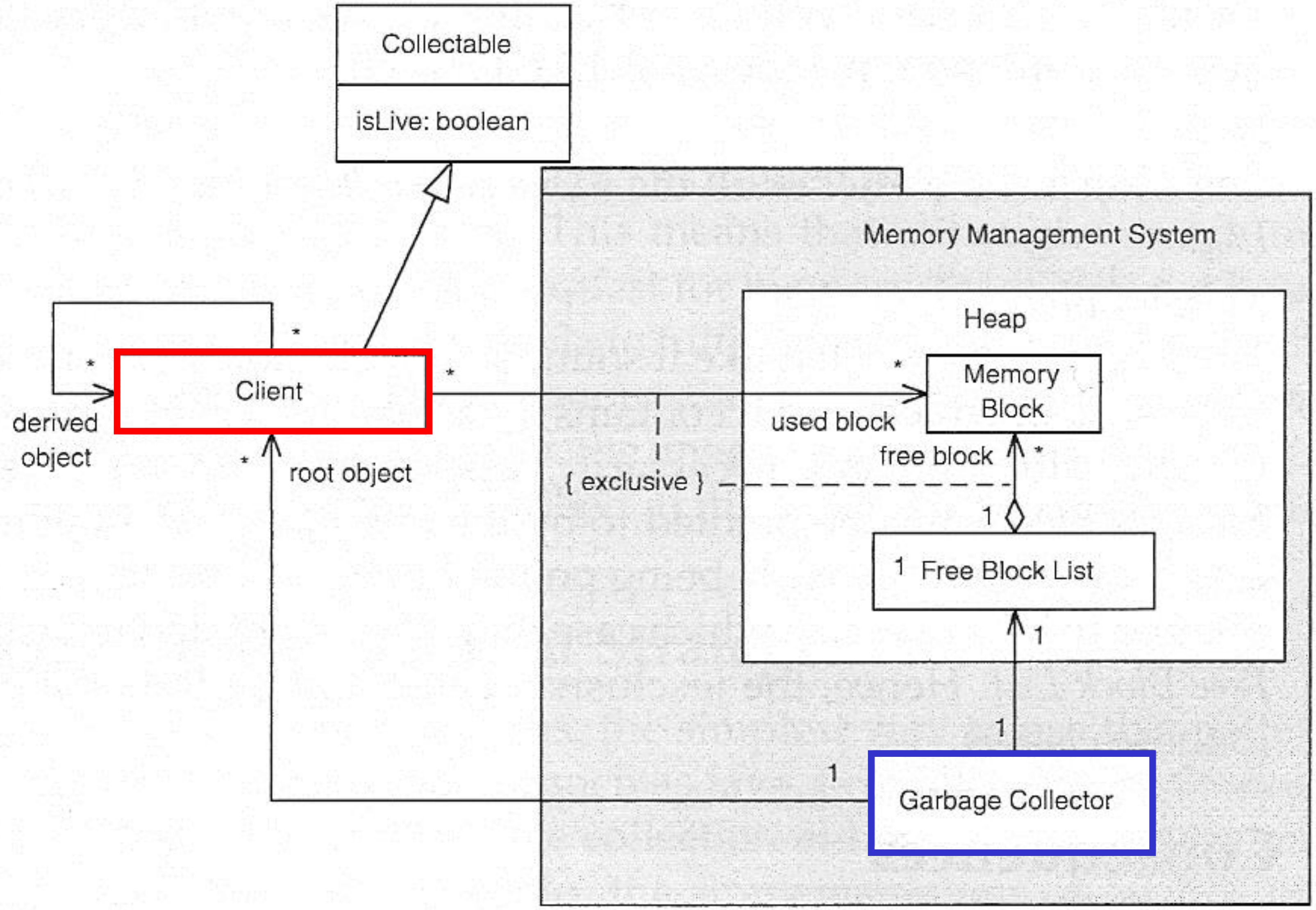
scoped_ptr	<boost/scoped_ptr.hpp>	Simple sole ownership of single objects. Noncopyable.
scoped_array	<boost/scoped_array.hpp>	Simple sole ownership of arrays. Noncopyable.
shared_ptr	<boost/shared_ptr.hpp>	Object ownership shared among multiple pointers.
shared_array	<boost/shared_array.hpp>	Array ownership shared among multiple pointers.
weak_ptr	<boost/weak_ptr.hpp>	Non-owning observers of an object owned by **shared_ptr**.
intrusive_ptr	<boost/intrusive_ptr.hpp>	Shared ownership of objects with an embedded reference count.

5. Garbage Collection Pattern

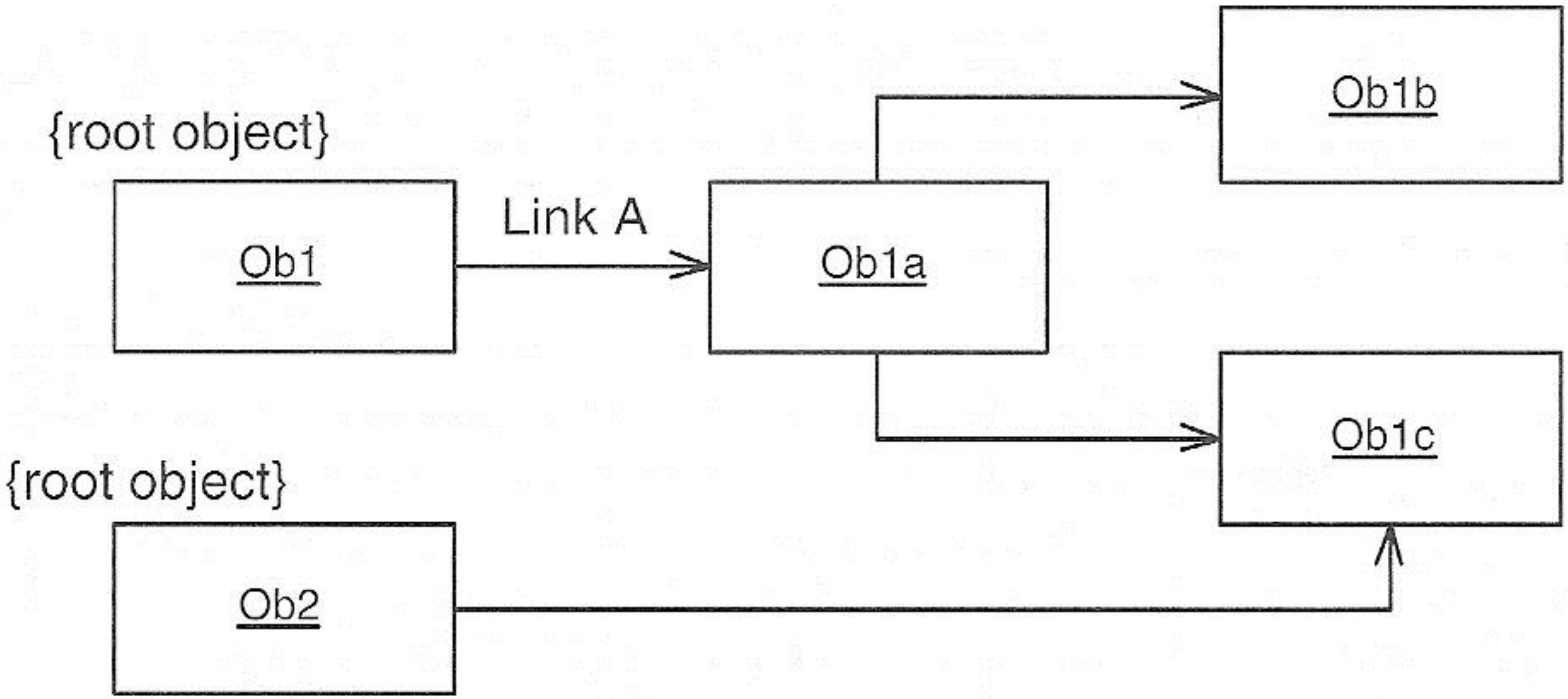
The Garbage Collection Pattern can eliminate memory leaks in programs that must use dynamic memory allocation

Note! Not applicable in hard real-time systems

Garbage Collection Pattern Structure

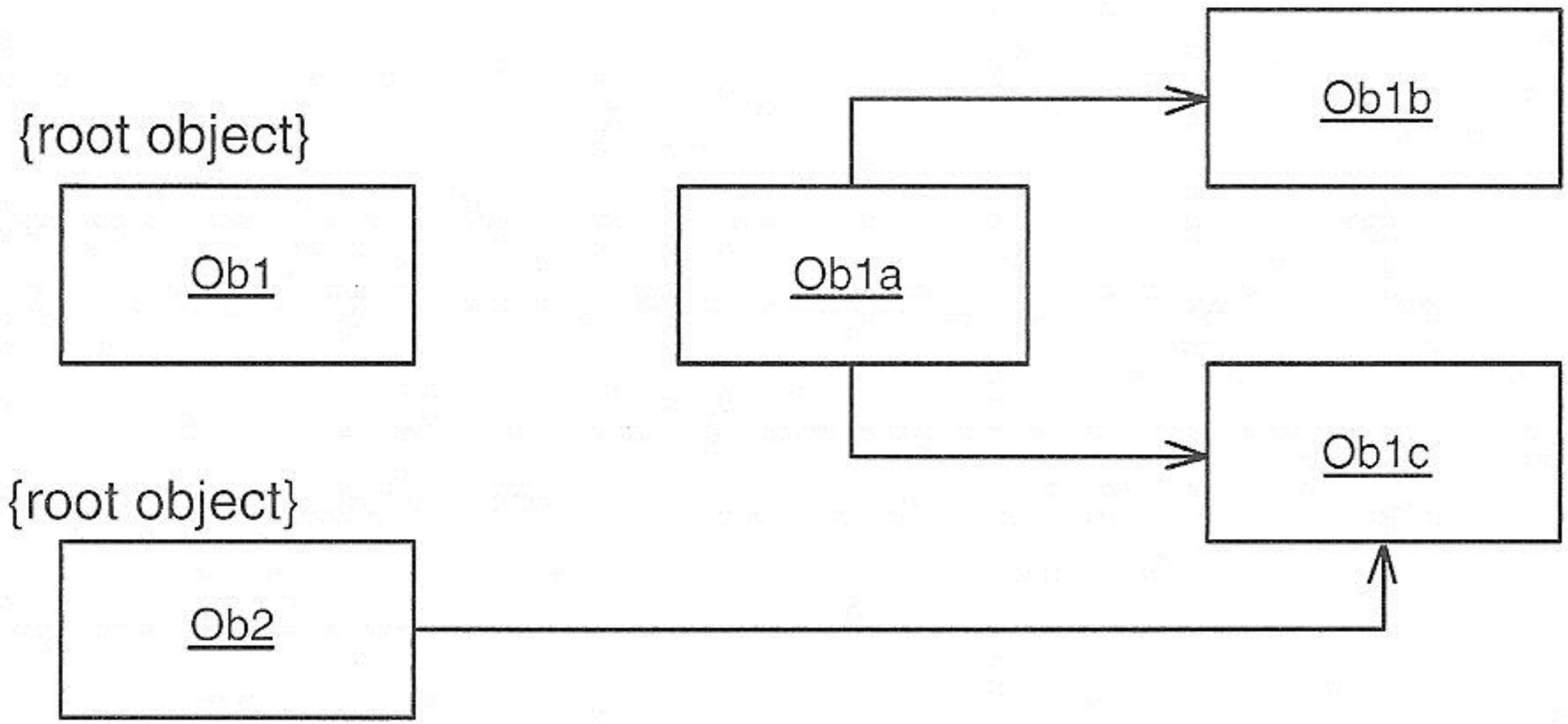


Garbage Collection Pattern (1)



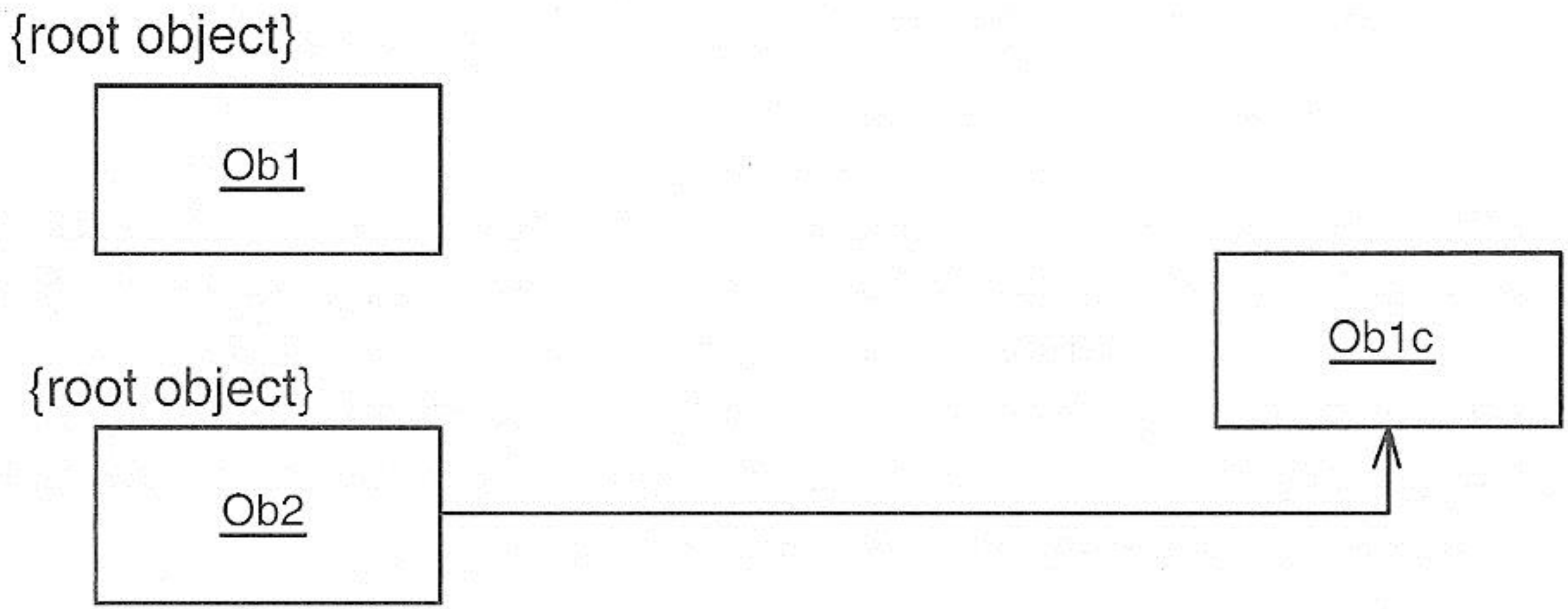
a. Instance Model Snapshot 1—Starting Instance Model

Garbage Collection Pattern (2)



b. Instance Model Snapshot 2—Before Garbage Collection

Garbage Collection Pattern (3)



c. Instance Model Snapshot 3—After Garbage Collection

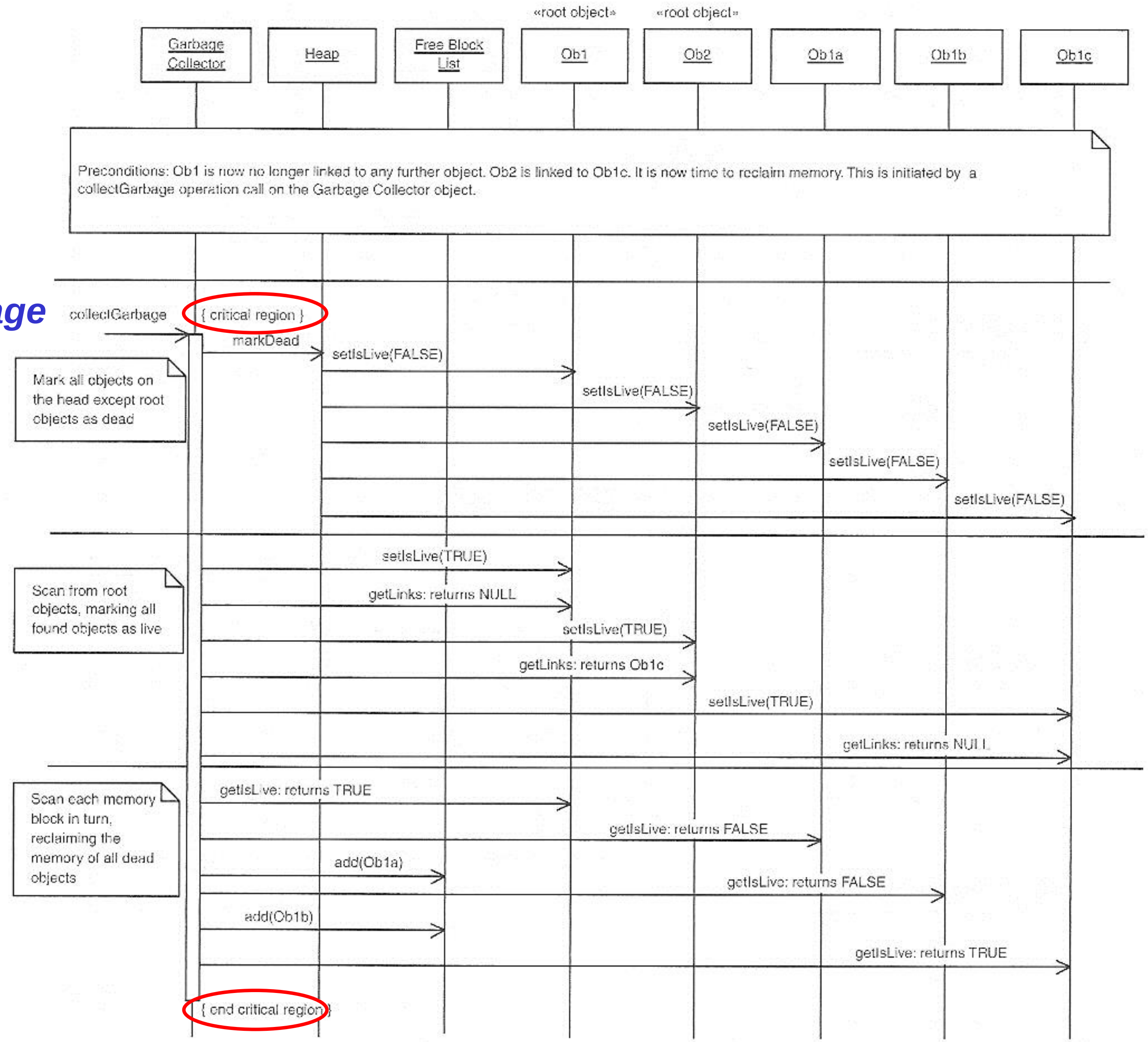
Garbage Collection Pattern Example

collectGarbage

1. Mark all dead

2. Mark all found as live

3. Reclaming all dead objects



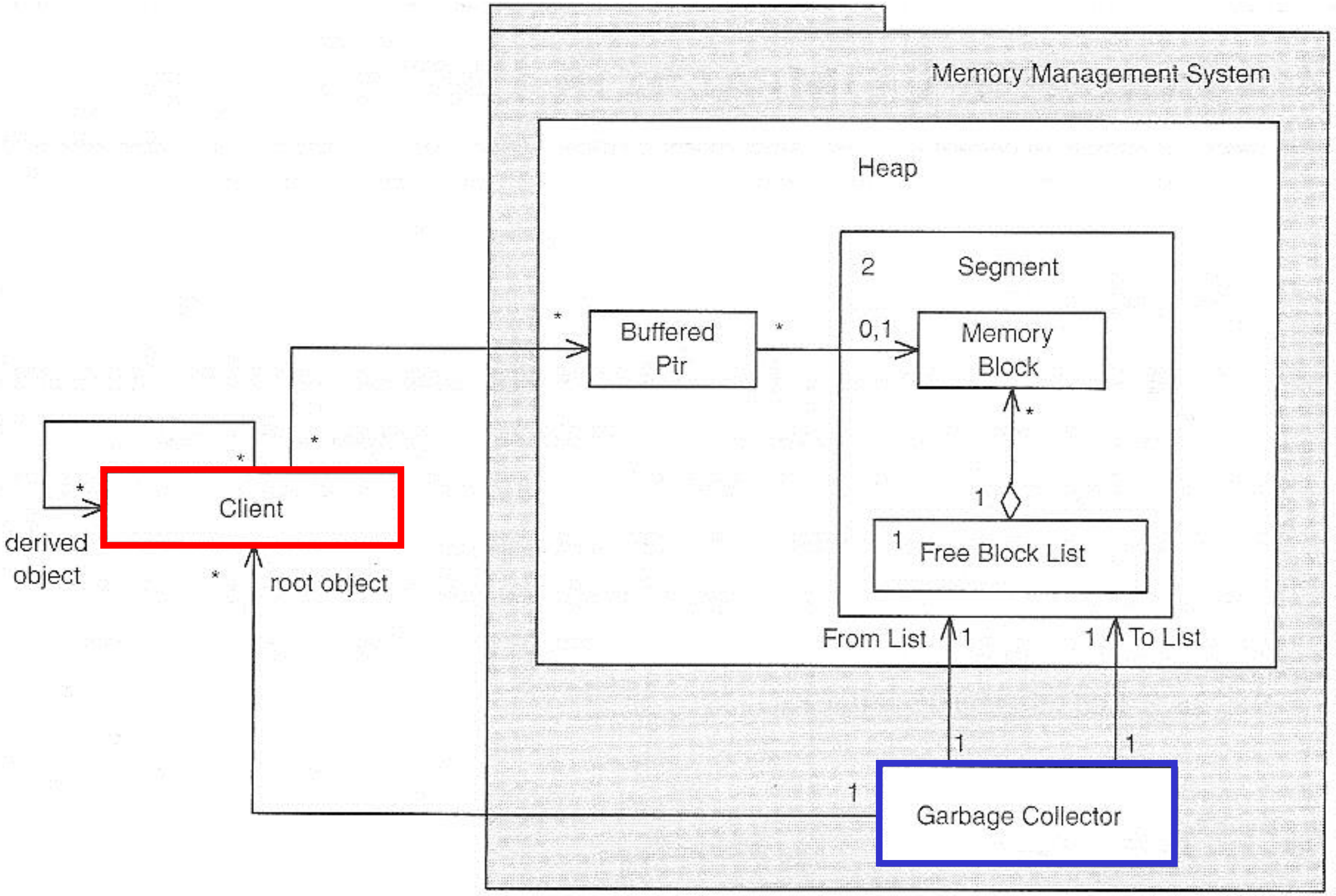
6. Garbage Compactor Pattern

The Garbage Compactor Pattern – a variant of the Garbage Collector Pattern that also removes memory fragmentation

Note! Not applicable in hard real-time systems

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Garbage Compactor Pattern Structure



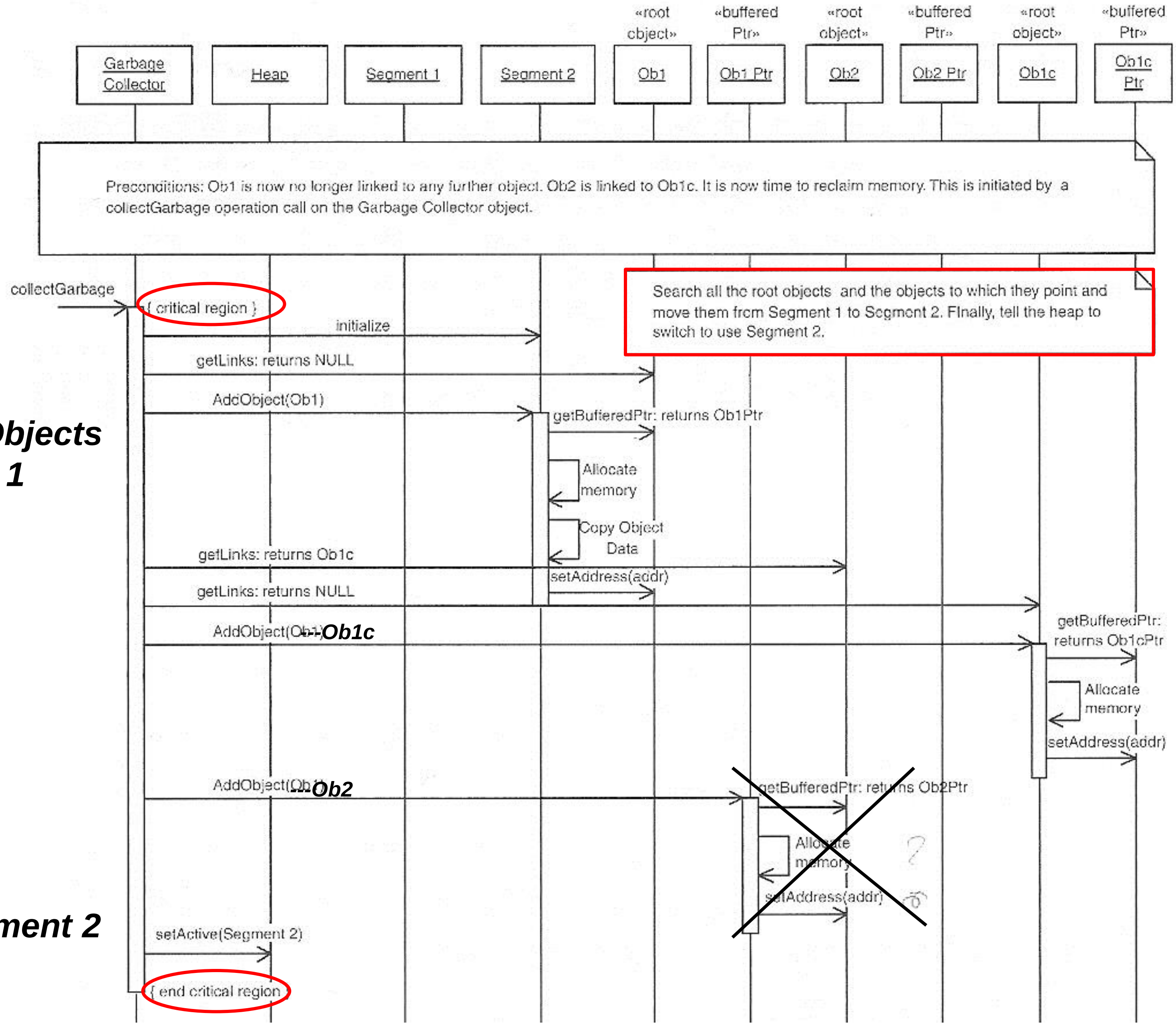
AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Garbage Compactor Pattern Example

collectGarbage

1. ***Move all ref. Objects from segment 1 to segment 2***

2. ***Switch to segment 2***



Summary

1. Static Allocation Pattern
2. Pool Allocation Pattern
3. Fixed Sized Buffer Pattern
4. Smart Pointer Pattern
5. Garbage Collection Pattern
6. Garbage Compactor Pattern