AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

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

POSA2: *Monitor Object Design Pattern + Traits*

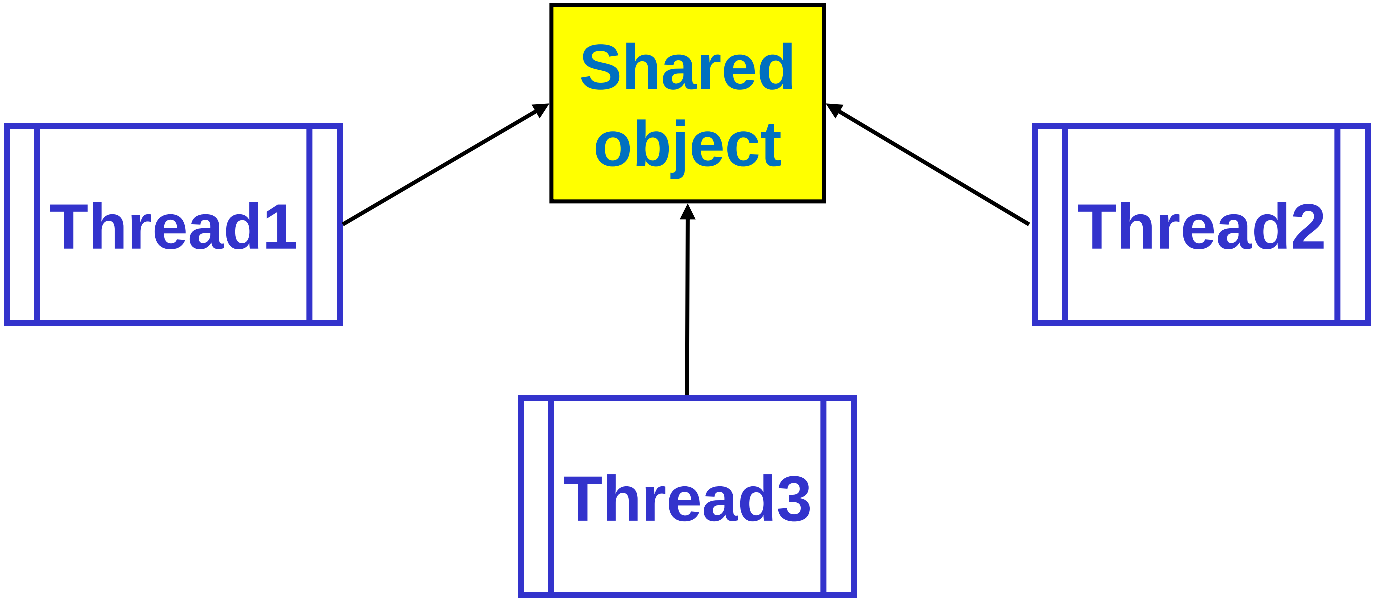
Version: 23-2-2015

Abstract

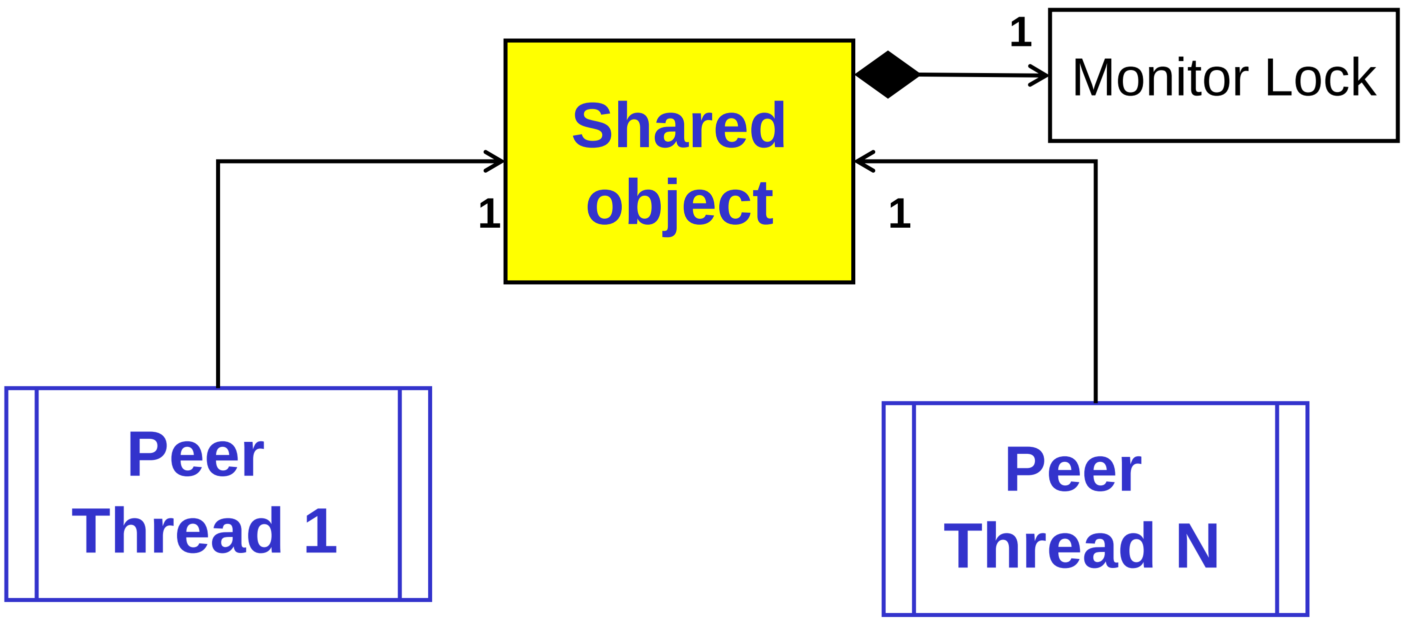
The Monitor Object design pattern **synchronizes concurrent method execution** to ensure that only one method at a time runs within an object.
It also allows an object's method to cooperatively schedule their execution sequences.

Monitor Object Context

- Multiple threads of control accessing the same object concurrently

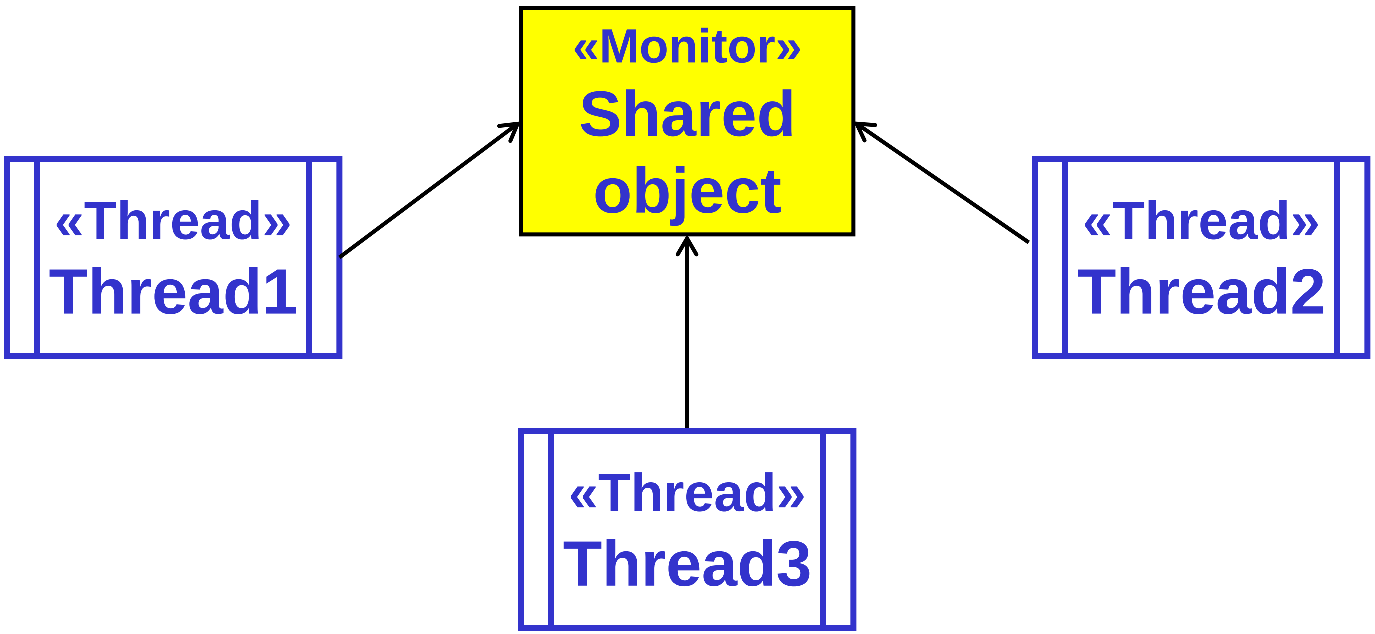


A Simple Monitor



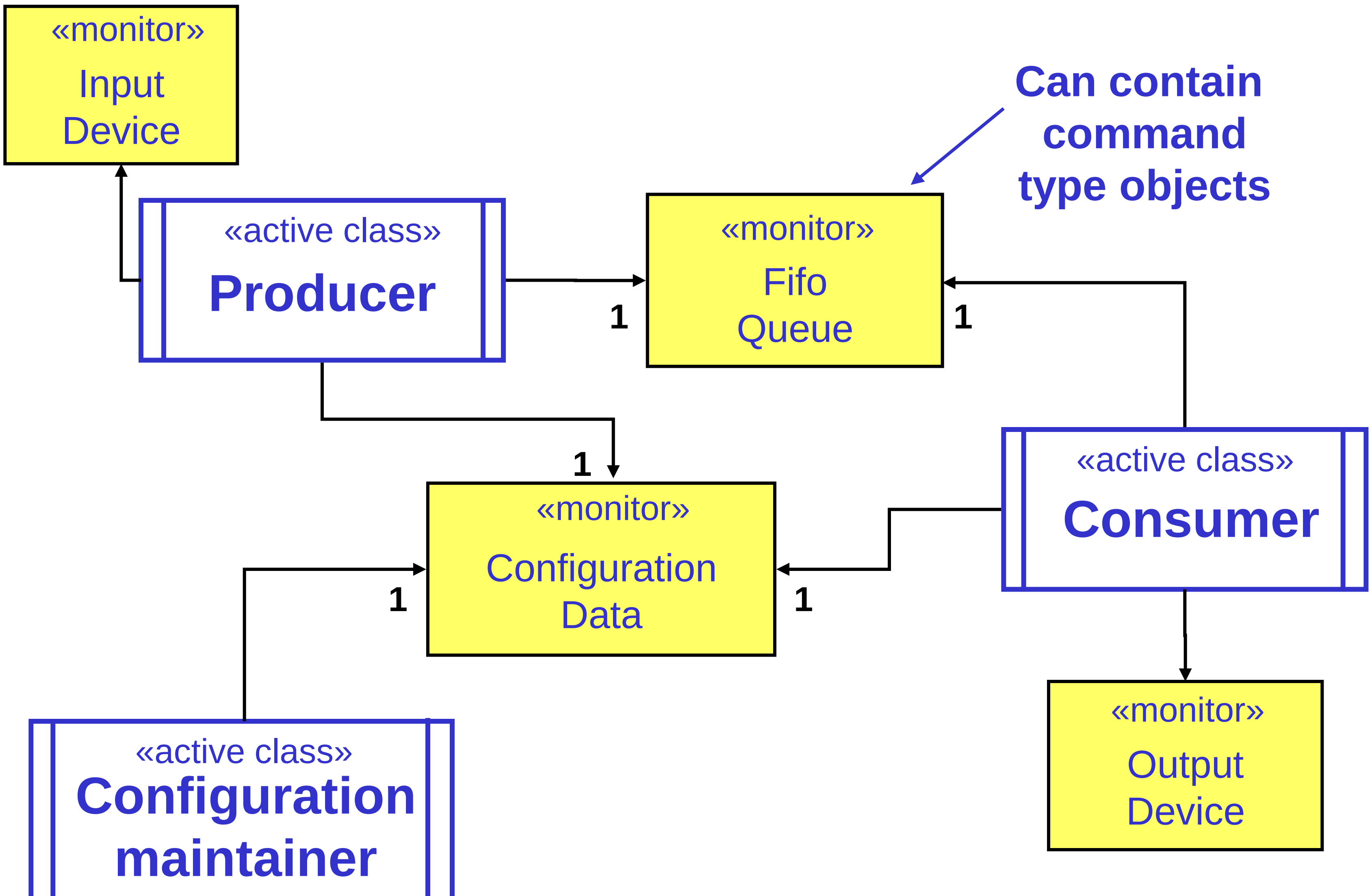
synchronous communication / call

UML Stereotype Annotations

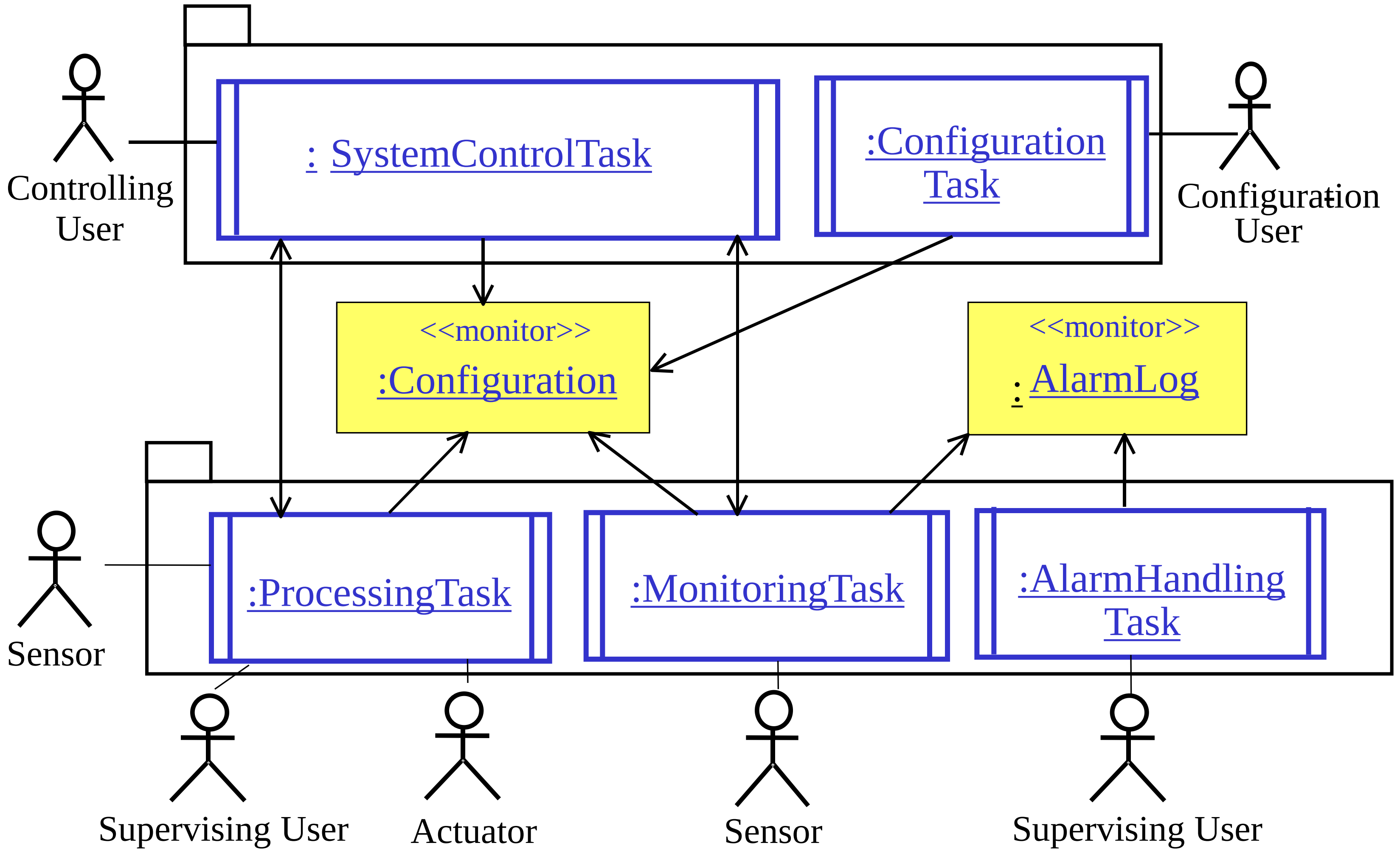


Stereotype used to indicate the pattern

Monitor Pattern Example (1)

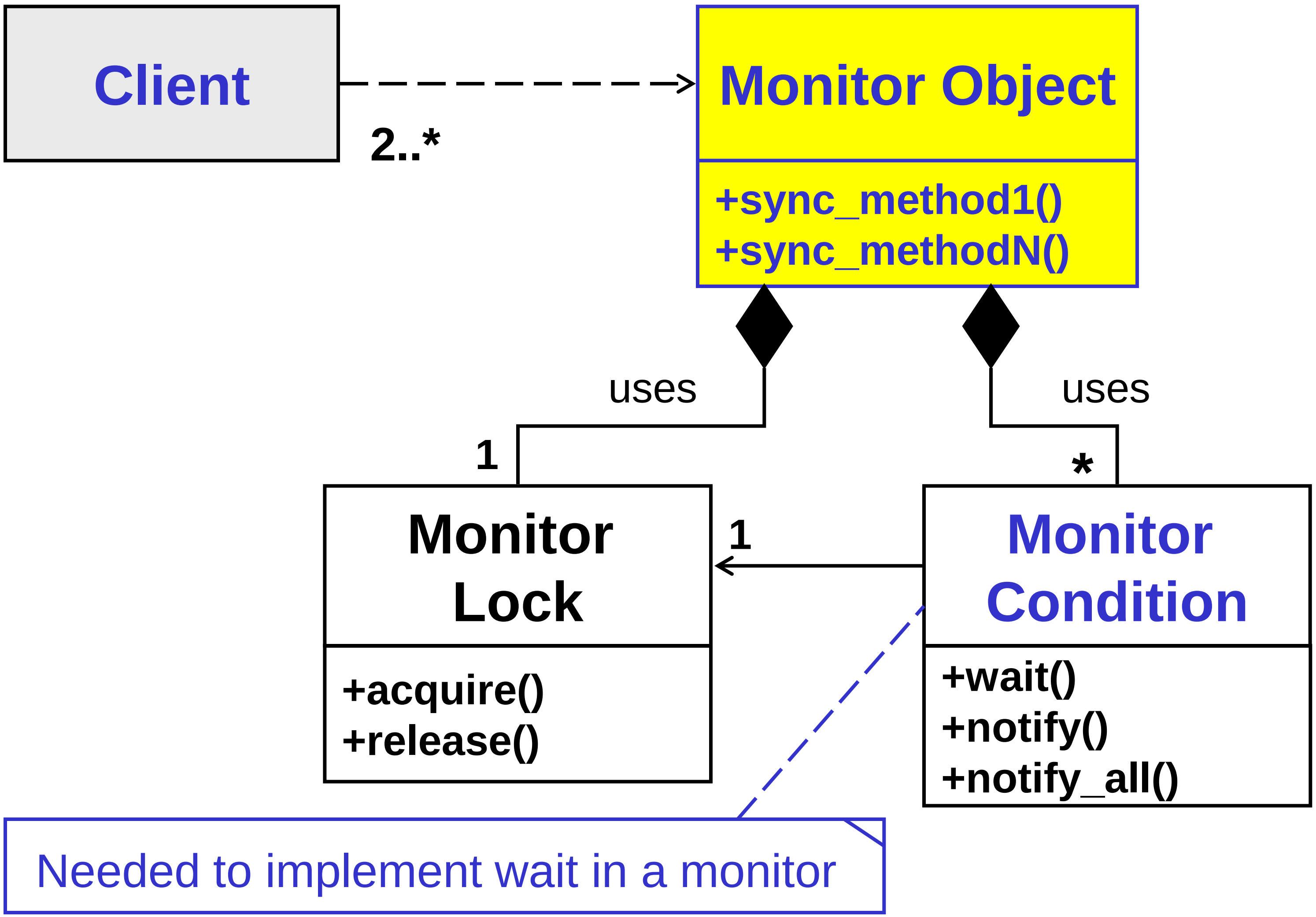


Monitor Pattern Example (2)



Two part architectural model

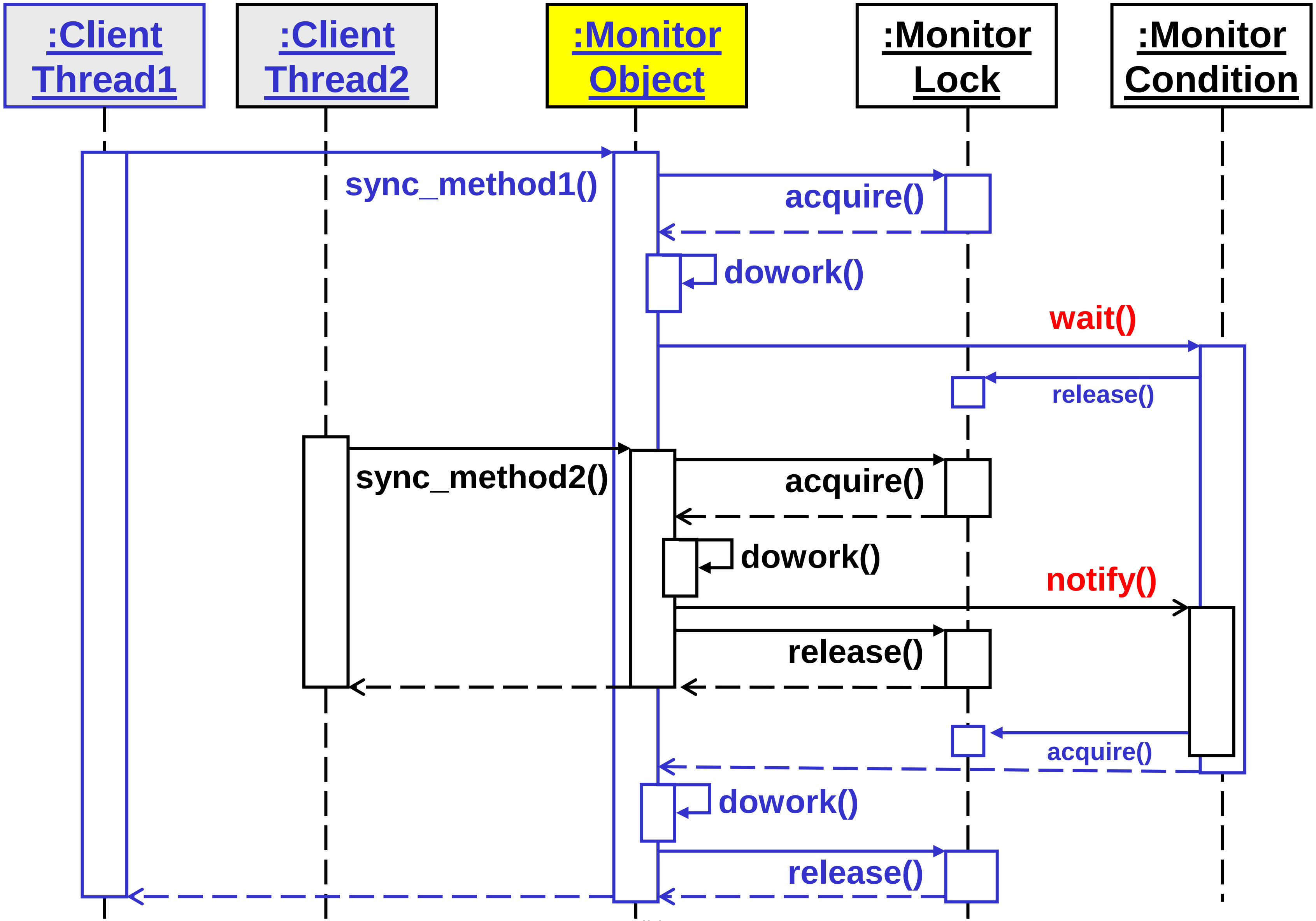
POSA2 Monitor Object Pattern Structure



Monitor Condition Variable

- Needed for implementing wait in a monitor
- Simplifies client thread programming and enhance efficiency
- Operation **”wait”** on a given condition and releases monitor lock
- Operation **“notify”** wakeup first waiting thread on the given condition
- Operation **“notify_all”** wakeup all waiting threads

Monitor Object Dynamics

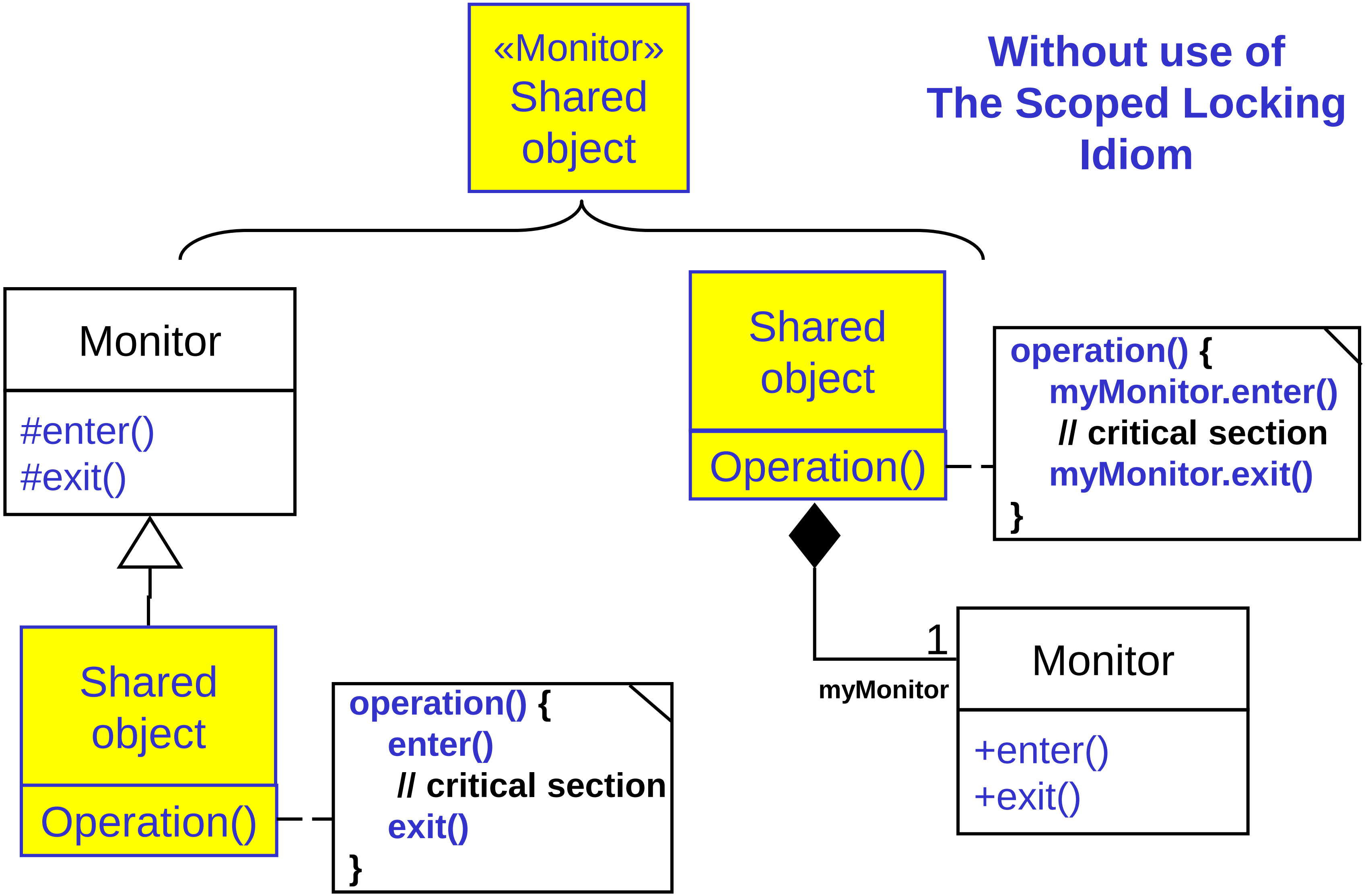


Monitor Implementation Steps

1. Define the monitor object’s interface methods
2. Define the monitor object’s implementation methods
3. Define the monitor object’s internal state and synchronization mechanism
4. Implement the monitor object’s methods and data members

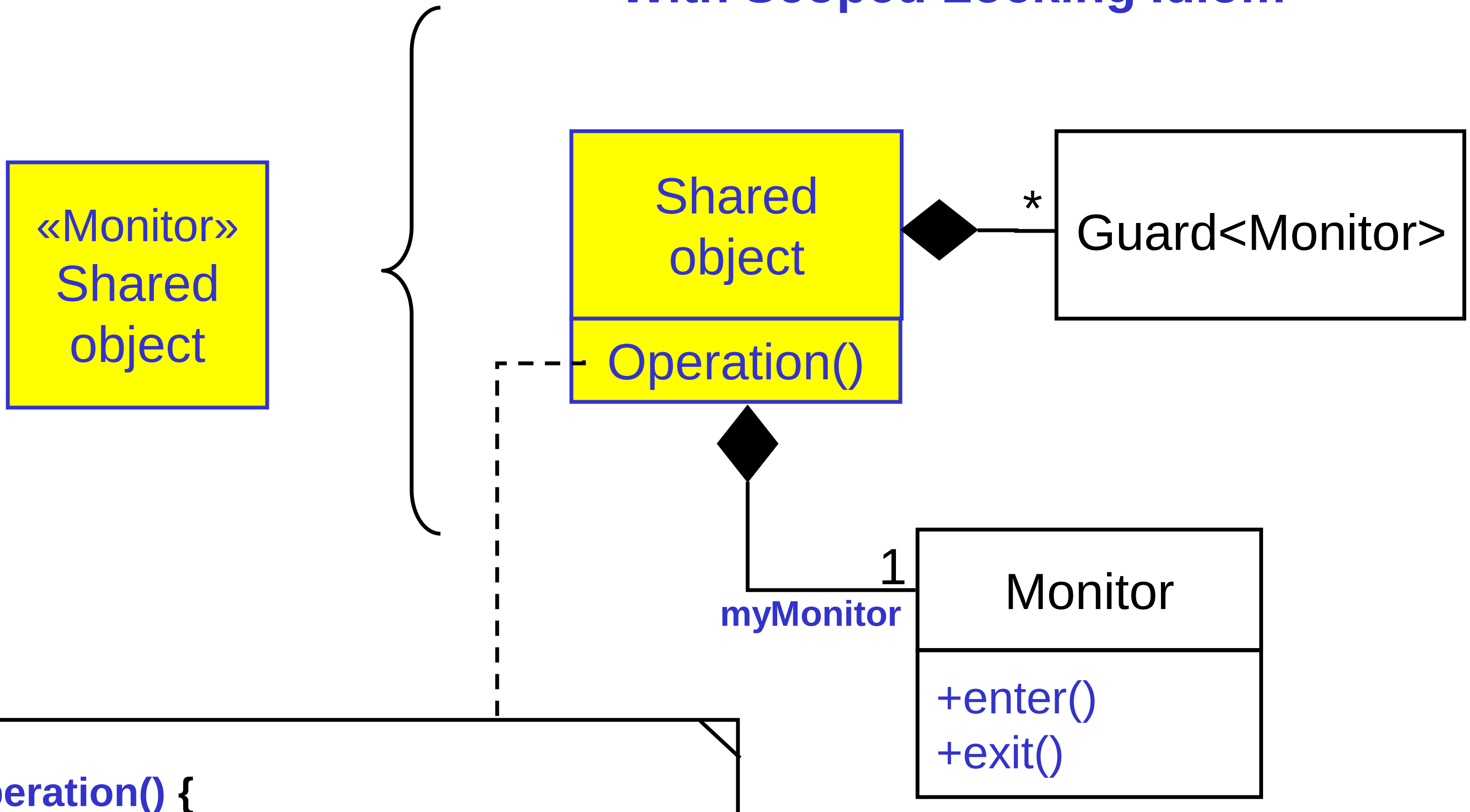
Monitor Implementation Strategies (1)

Without use of The Scoped Locking Idiom



Monitor Implementation Strategies (2)

With Scoped Locking Idiom



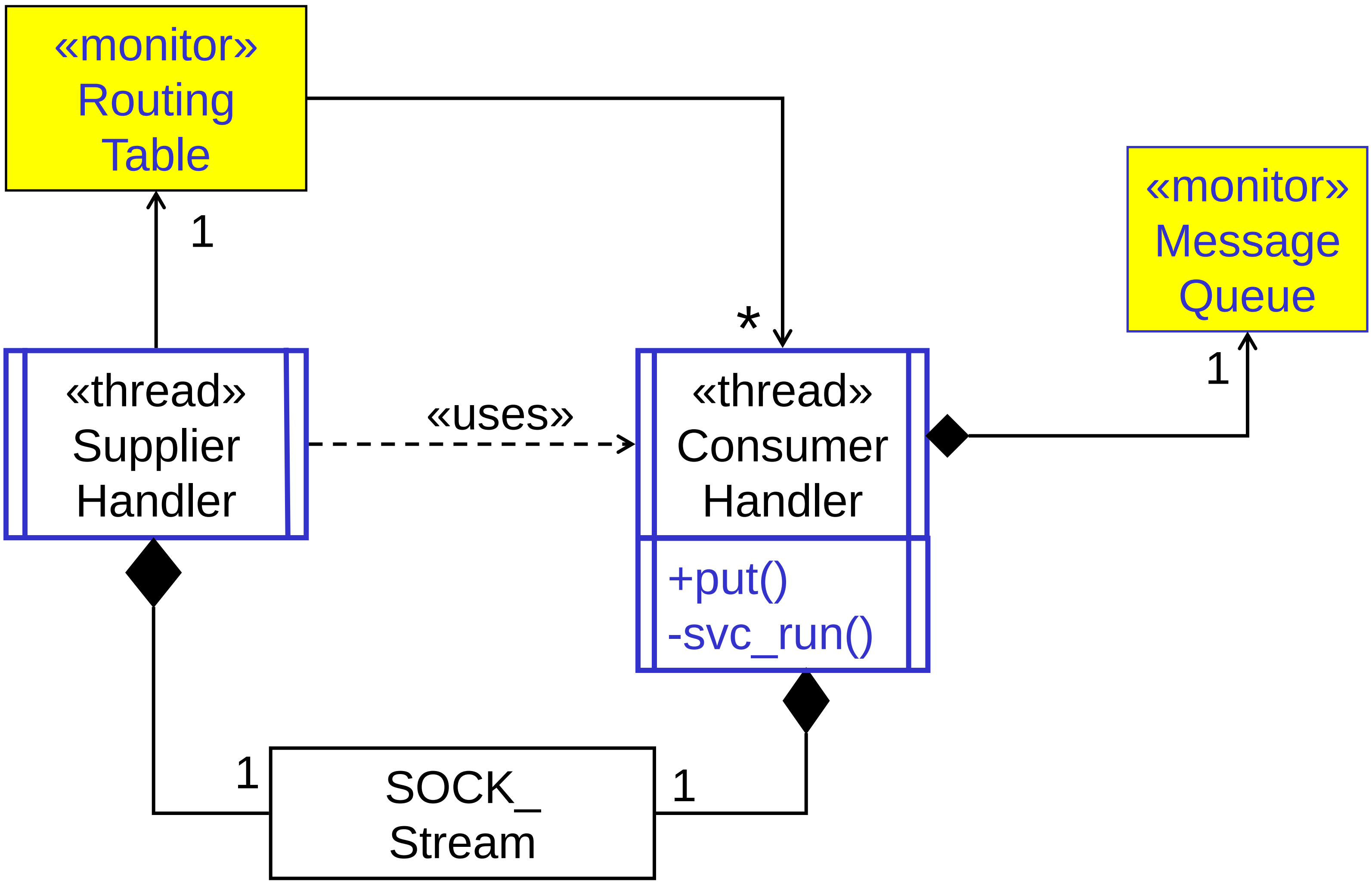

```
operation() {
   Guard<Monitor> guard(&myMonitor);
    // critical section
}
```

Solution: Scoped Locking C++ Idiom

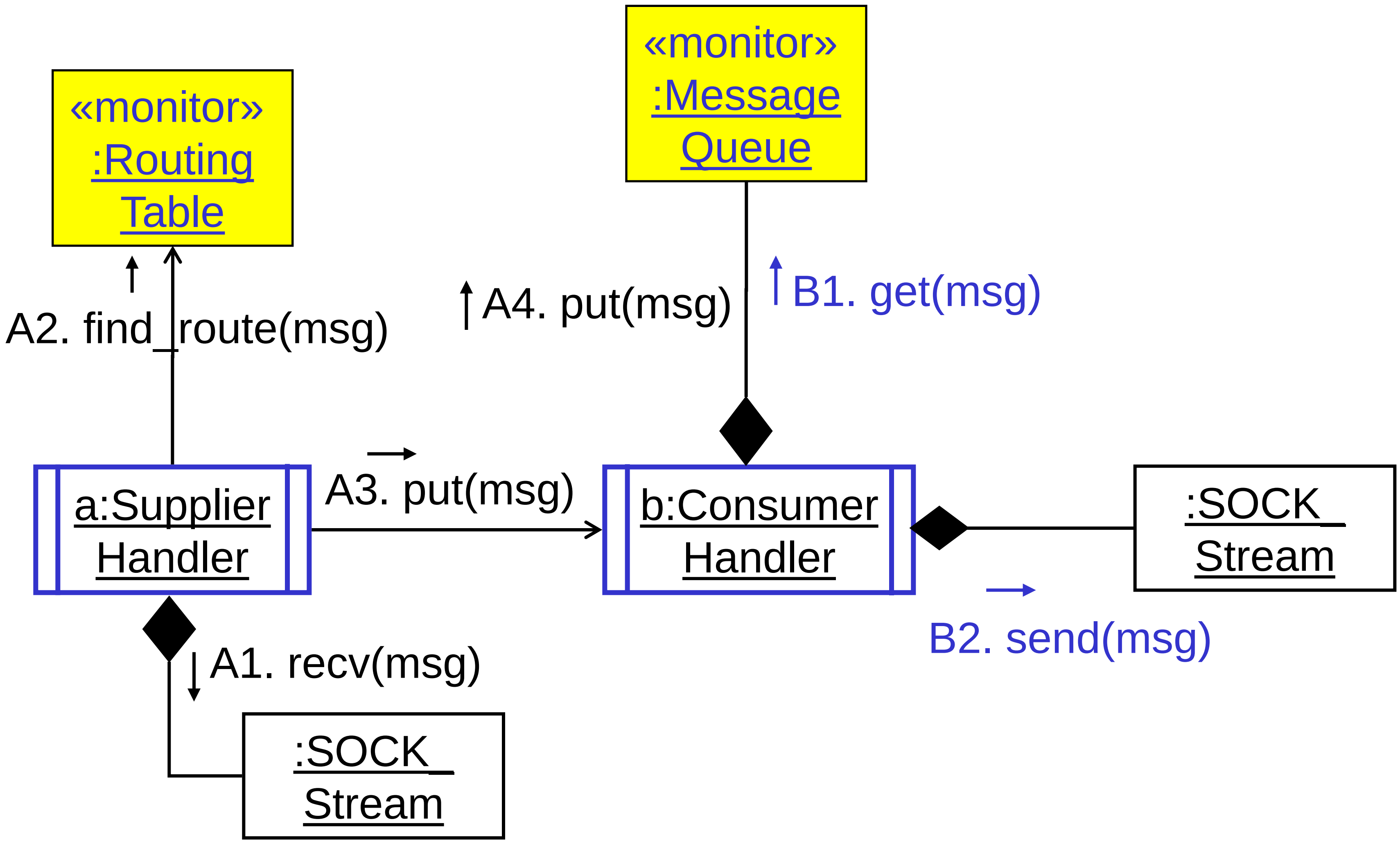
- Define a **guard class** whose **constructor** acquires a lock when control enters a scope and whose **destructor** automatically releases the lock when control leaves the scope

```
void MonitorClassX::operationX()
{
   Guard myGuard(lock_);    // lock_ pointer to a lock object
    // …  Critical section code
    // …
    return;
}    // destructor called automatically
```

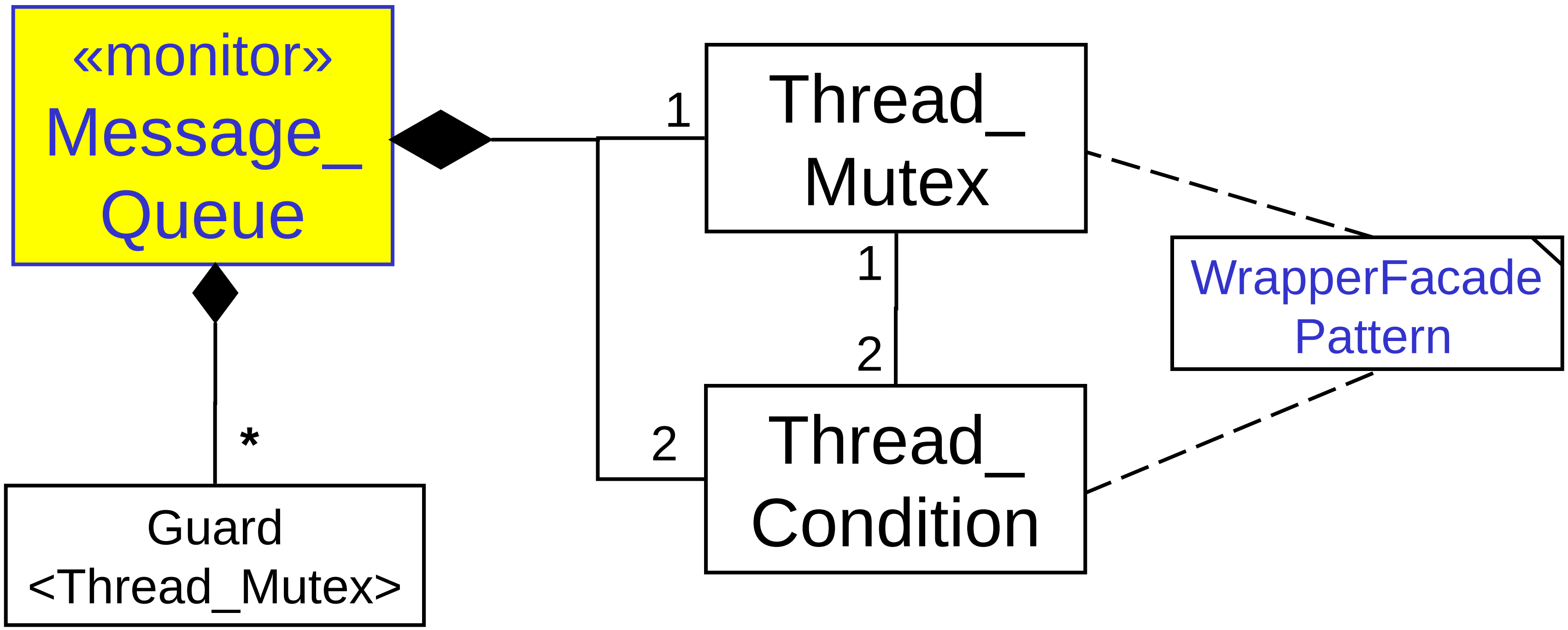
Gateway Example – Class Diagram



Gateway Example - Collaboration



Message_Queue Monitor Class



1. Define the monitor object's interface methods

«monitor» Message_ Queue	
+void put(Message &msg)	// synchronized
+Message get();	// synchronized
+bool empty();	// synchronized
+bool full();	// synchronized

2. Define the monitor object's implementation methods

«monitor»
Message_
Queue

+void put(Message &msg)
+Message get();
+bool empty();
+bool full();
-void put_i(Message &msg)
-Message get_i();
-bool empty_i();
-bool full_i();

Example of:
POSA2 Thread-Safe Interface Pattern

Interface methods only acquire and release monitor locks

They forward control to **private** implementations methods that perform the monitor object's functionality

Class Message_Queue

```
class Message_Queue {
public:
   enum { MAX_MESSAGES= 100; };
   Message_Queue(size_t max_messages= MAX_MESSAGES);
   void put(const Message &msg);          // Synchronized
   Message get();                         // Synchronized
   bool empty() const;                    // Synchronized
   bool full() const;                     // Synchronized
private:
   void put_i(const Message &msg);        // non synchronized
   Message get_i();
   bool empty_i() const;
   bool full_i() const;
   size_t message_count_, max_messages;
   mutable Thread_Mutex monitor_lock_;
   Thread_Condition not_empty_;           } Two condition
   Thread_Condition not_full_;            }   variables
};
```

Thread_Mutex Example (Solaris)

```
class Thread_Mutex {            // OS wrapper class
public:
   Thread_Mutex()    { mutex_init(&mutex_,USYNC_THREAD,0); }
   ~Thread_Mutex()   { mutex_destroy(&mutex_); }
   void acquire()    { mutex_lock(&mutex_); }
   void release()    { mutex_unlock(&mutex_); }
private:
   // Solaris-specific Mutex mechanism
   mutex_t mutex_;

   // disallow copying and assignment
   Thread_Mutex(const Thread_Mutex &);
   void operator=(const Thread_Mutex &);
   friend class Thread_Condition;
};
```

Tread_Condition Example (Solaris)

```
class Thread_Condition {          // OS wrapper class
public:
   Thread_Condition(const Thread_Mutex &m) : mutex_(m)
         { cond_init(&cond_, USYNC_THREAD, 0); }
   ~Thread_Condition() { cond_destroy(&cond_); }
   void wait(Time_Value *timeout= 0) {
      cond_timedwait(&cond_, &mutex_.mutex_,
                        timeout==0 ? 0 : timeout->msec());
   }
   void notify() { cond_signal(&cond_); }
   void notify_all() { cond_broadcast(&cond_); }
private:
   condt_t cond_;   // SOLARIS condition variable
   const Thread_Mutex &mutex_;
};
```

Thread_Mutex_Guard Class

```
class Thread_Mutex_Guard {
public:
   Thread_Mutex_Guard(Thread_Mutex &lock)
                              : lock_(&lock), owner_(false) { acquire(); }
   void acquire() {lock_->acquire(); owner_=true; }
   void release() {
      if (owner_) {
           owner_=false; lock_->release();  }
   }
   ~Thread_Mutex_Guard() { release(); }
private:
   Thread_Mutex *lock_;  // Pointer to our lock
   bool owner_;               // is lock held by this object ?

    // disallowing copy and assignment
   Thread_Mutex_Guard(const Thread_Mutex_Guard &);
   void operator=(const Thread_Mutex_Guard &);
};
```

Guard Template Class

```
template <class LOCK>
class Guard {
public:
    Guard(LOCK &lock) : lock_(&lock), owner_(false) { acquire(); }
    void release() {
        if (owner_) {
              owner_=false; lock_->release();  }
    }
    ~Guard() { release(); }
protected:
    void acquire() {lock_->acquire(); owner_=true; }
private:
    LOCK *lock_;  // Pointer to our lock
    bool owner_;                  // is lock held by this object ?
     // disallowing copy and assignment
    Guard(const Guard &);
   void operator=(const Guard &);
 // ….
};
```

Class Message_Queue Code

```
Message_Queue::Message_Queue(size_t max_messages)
            : not_full_(monitor_lock_), not_empty_(monitor_lock_),
             max_messages_(max_messages), message_count_(0) { }

bool Message_Queue::full() const {
   Guard<Thread_Mutex> guard(monitor_lock_);
   return full_i();
}

void Message_Queue::put(const Message &msg) {
   Guard<Thread_Mutex> guard(monitor_lock_);        // see page 337
   while (full_i()) {
      // Release <monitor_lock_> and suspend calling thread
      // the monitor lock is reacquired automatically when <wait> returns
      not_full_.wait();
   }
   put_i(msg);
   not_empty_.notify();
} // destructor of <guard> releases <monitor_lock_>
```

Supplier_Handler::route_message

```
void Supplier_Handler::route_message(const Message &msg)
{
   // Locate the appropriate consumer based on the address info in msg
   Consumer_Handler *consumer_handler_=
          routing_table_.find_route(msg.address());

   // Put the Message into the Consumer Handler's queue
   consumer_handler_->put(msg);
}
```

Notice: only application level programming (no OS locking mechanism)

Class Consumer_Handler

```
class Consumer_Handler {      // THREAD
public:
  Consumer_Handler()
          { Thread_Manager::instance()->spawn(&svc_run, this); }
  void put(const Message &msg) { message_queue_.put(msg); }

private:
  Message_Queue message_queue_;      // Monitor Object
  SOCK_Stream connection_;

  static void *svc_run(void *args) {
    Consumer_Handler *this_obj= static_cast<Consumer_Handler *> (args);
    for (; ;) {
      // Block on <get> until next message arrives
      Message msg= this_obj->message_queue_.get();
      this_obj->connection_.send(msg, msg.length());
    }
  }
};
```

Variants: Strategized Locking

```
template <class SYNC_STRATEGY>        // C++ TRAITS EXAMPLE
class Message_Queue {
        //
private:
   typename SYNC_STRATEGY::Mutex monitor_lock_;
   typename SYNC_STRATEGY::Condition not_empty_;
   typename SYNC_STRATEGY::Condition not_full;
}

// Example of an implementation method: empty()

template <class SYNC_STRATEGY>
bool Message_Queue< SYNC_STRATEGY>::empty() const {
   Guard<SYNC_STRATEGY::Mutex> guard(monitor_lock_);
   return empty_i();
}
```

Synchronization Traits (1)

```
class MT_Synch {
public:
    // Synchronization traits
    typedef Thread_Mutex Mutex;
    typedef Thread_Condition Condition;
};
```

**Definition of a thread-safe Message_Queue,
using Thread_Mutex and Thread_Condition wrapper classes:**

```
Message_Queue<MT_Synch> message_queue;
```

Synchronization Traits (2)

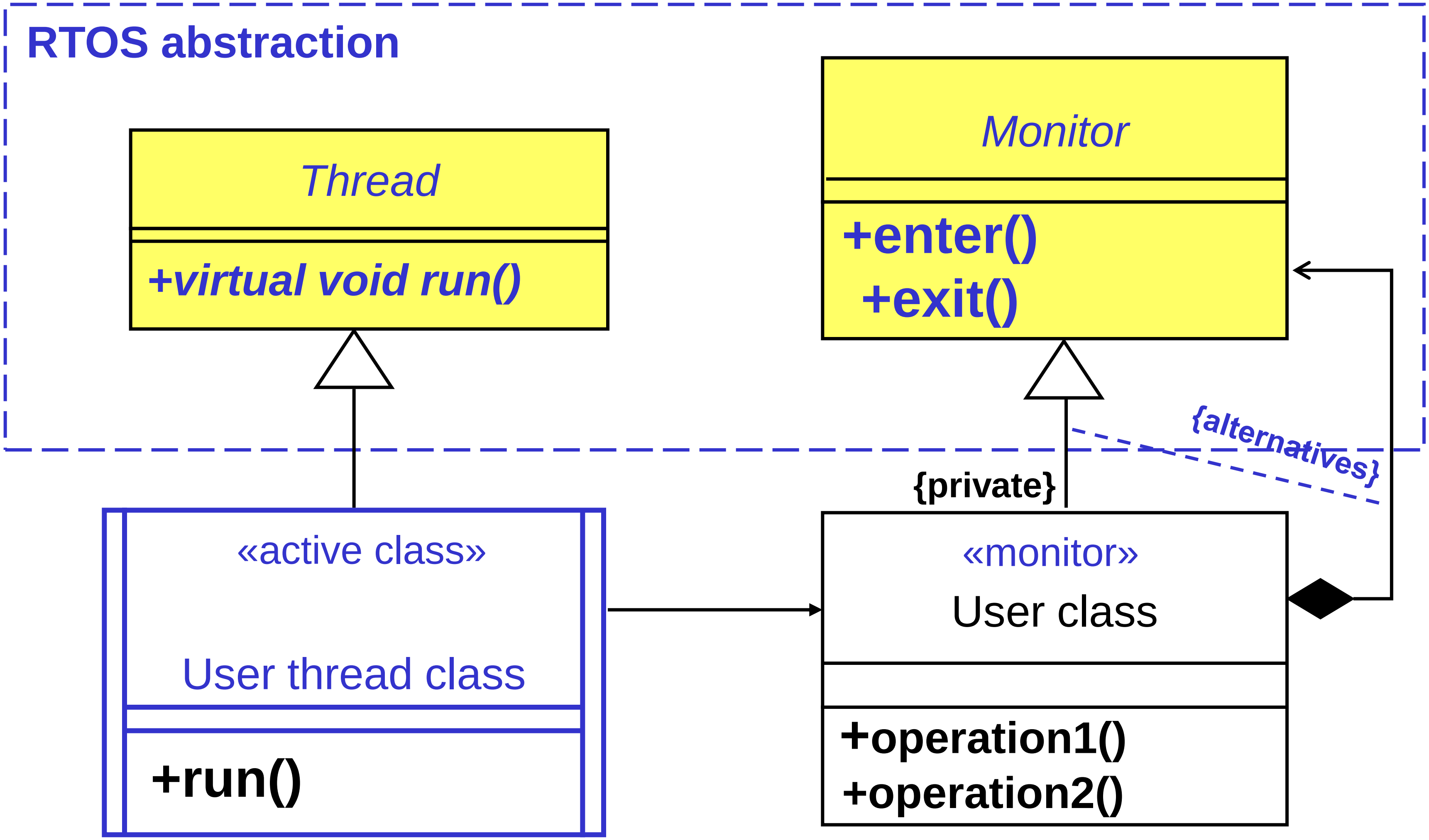
```
class Null_Synch {
public:
   // Synchronization traits
   typedef Null_Mutex Mutex;
   typedef Null_Condition Condition;
};
```

```
class Null_Mutex {
public:
   Null_Mutex() { }
   ~Null_Mutex() { }
   void acquire() { }
   void release() { }
};
```

Definition of a non-thread-safe Message_Queue,

Message_Queue<Null_Synch> message_queue;

Two useful OS Abstractions



Monitor Object Benefits

- Simplification of Concurrency control
  - clients need not to be concerned with concurrency control
- Simplification of scheduling method execution
  - using monitor condition objects

Monitor Object Liabilities

- The use of a single monitor lock can limit scalability due to increased contention
- Complicated extensibility semantics due to a tight coupling between functionality and synchronization mechanism
- It is hard to inherit from a monitor
- Nested monitor lockout, when a monitor is nested in another monitor

Known Uses

- Dijkstra, Hoare & Brinch Hansen Monitors
- Java Objects (synchronized)
- ACE Gateway

Relation to other POSA2 Patterns