Methodical Concurrency Design in Education Part two: Deadlock Technical Report (TR-OU-INF-2015-01b)

A. Bijlsma[*1], C. Bockisch[†1], H.J.M. Passier[‡1] and H.J. Pootjes[§1]

[1] *Open Universiteit, Faculty of Management, Science and Technology, Department of Computer Science, Postbus 2960, 6401 DL Heerlen, The Netherlands*

December 17, 2015

[*] lex.bijlsma@ou.nl
[†] christoph.Bockisch@ou.nl
[‡] harrie.passier@ou.nl
[§] harold.pootjes@ou.nl

1 Introduction

The term concurrent programming refers to the development of software with several threads of execution that are performed (logically) at the same time. These threads will not always be executed at the same relative speed and it is therefore non-deterministic which instructions are actually performed concurrently. As a consequence, so-called race conditions can occur: The program's concrete outcome depends on the timing of the concurrent threads. Concretely the access to resources (often data) that are shared between threads must therefore be organized such that threads do not interfere with each other. The general goal is that a concurrently executing thread gets the same result when accessing a resource as if the thread was the only one executing.

To achieve this effect, developers of concurrent programs must apply synchronization in their code, as we have detailed in part one [3]: A thread must acquire an exclusive lock before accessing a critical resource and release it afterwards. Concurrently executing threads that want to acquire a lock already in use must wait until it is released. But applying synchronization can cause lock-ordering deadlocks: The simplest case of deadlock is a thread *t1* holding lock *lock_1* and trying to acquire lock *lock_2*, while at the same time a thread *t2* holds lock *lock_2* and tries to acquire *lock_1*, in which both threads will wait forever (in section 3.2 we introduce a diagram notation to depict and analyse such situations). As a result of deadlock the threads waiting for each other are blocked and the program cannot fulfill its task and terminate normally. Because Java programs do not recover autonomously from deadlock, and most operating systems do not either, it is advisable to ensure that a program cannot cause deadlocks.

Listing 1: Example of potentially deadlocking code.

```
final Object lock_1 = new Object();
final Object lock_2 = new Object();

Thread t1 = new Thread(new Runnable() {
  public void run() {
    synchronized (lock_1) {
      synchronized (lock_2) {
        System.out.println("do something");
      }
    }
  }
});

Thread t2 = new Thread(new Runnable() {
  public void run() {
    synchronized (lock_2) {
      synchronized (lock_1) {
```

```
            System.out.println("do something else");
        }
      }
    }
});

t1.start();
t2.start();
```

As an example, consider the listing above. The two threads *t1* and *t2* are executed concurrently, but we cannot be sure about the actual timing of the execution of the threads statements. One possibility is that thread *t1* is executed completely before *t2* starts: In the beginning both locks are free. Thread *t1* acquires *lock_1*, then *lock_2* and accesses a critical resource (in this case represented by printing the string "do something"). Afterwards the thread releases first *lock_2* and then *lock_1*. Therefore, both locks are free when thread *t2* begins execution and no deadlock occurs.

But the threads' execution may also be scheduled such that their instructions are really executed in parallel or they are interleaved (i.e., statements from both threads are executed alternatingly). As an example consider such an interleaving illustrated by Table 1 below. The two left-most columns represent the threads and the rows show the statements of the thread being executed, where each row represents one timestamp. The two right-most columns show the locks and by which thread they are acquired after the execution of the statements in a row.

Thread *t1*	Thread *t2*	Lock *lock_1*	Lock *lock_2*
		free	free
synchronized (*lock_1*) → *try to lock lock_1*		*t1*	*t2*
	synchronized (*lock_2*) → *try to lock lock_2*	*t1*	*t2*
synchronized (*lock_2*) → *wait for t2 to release lock_2*		*t1*	*t2*
waiting	**synchronized** (*lock_1*) → *wait for t1 to release lock_1*	*t1*	*t2*
waiting	*waiting*	*t1*	*t2*

Table 1: An example of thread interleaving resulting in deadlock

Figure 1 shows the same situation in a diagram. In this diagram, both threads *t1* and *t2* run from left to right and each line represents the execution of the activities of a thread in time.

Several dynamic tools are available to analyze deadlock situations *after they occur*

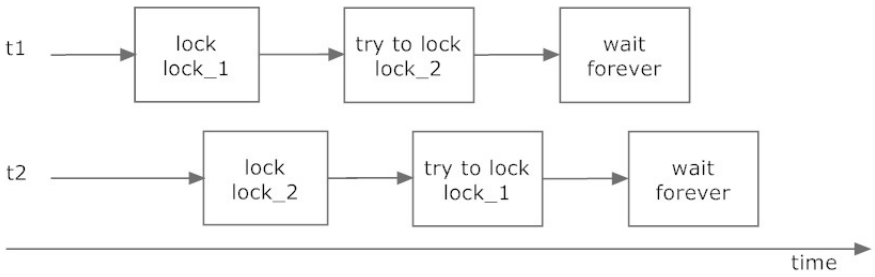


Figure 1: A depiction of thread interleaving resulting in deadlock

at runtime. They allow for example to inspect for which lock a thread is currently waiting and by which other thread the lock is held. The JDK contains such a tool: JVisualVM. Instead of using a tool you can also use class *ThreadMXBean* in your code to be able to analyze a thread dump.

Static tools that can signal potential deadlock problems are scarce. The well-known tool FindBugs[1] can indicate some troublesome concurrency aspects like calling *Thread.sleep* while holding a lock, but cannot indicate a potential deadlock. The commercial tool ThreadSafe[2] can only indicate an obvious case of deadlock like in the first listing, but not for more hidden cases like method transferMoney, see section 4.4.

In the following sections, we describe a procedure to determine *beforehand* if a deadlock could arise, to analyze these situations, and how we can *prevent* deadlocks. We also will look at dynamic tools that can indicate a potential deadlock.

[1]See homepage of FindBugs: `http://findbugs.sourceforge.net`
[2]See homepage of ThreadSafe: `http://www.contemplateltd.com/threadsafe`

2 Approach

Generally, there are four necessary conditions for deadlock [6, 9]:

Mutual exclusion. At least one lock must be held in a non-sharable mode, i.e. only one thread at a time can use it. If another thread requests that lock, the requesting thread must be delayed until the lock has been released.

Hold and wait. A thread must be holding at least one lock and waiting to acquire additional locks that are currently being held by other threads.

No preemption. Locks cannot be preempted, i.e. a lock can be released only voluntary by the thread holding it, after it has completed its task.

Circular wait. A set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads must exist such that T_0 is waiting for a lock held by T_1, T_1 is waiting for a lock held by T_2, ..., T_{n-1} is waiting for a lock held by T_n, and T_n is waiting for a lock held by T_0.

All four conditions must hold for a deadlock to occur. Thus, to determine whether a program can cause deadlock or not we must identify whether all four conditions may hold at once.

The Java language only knows *non-preemptive*, *mutual exclusion* locks which are always acquired through the **synchronized** keyword.[3] Every object in Java has a unique associated lock which can be acquired through this keyword.

There are two forms of this keyword: a statement and a method modifier. The statement **synchronized**(<*expression*>) {<*block*>} consists of an expression and a block. The expression is evaluated to an object whose associated lock is to be acquired. The block is executed after the lock is acquired and after the execution of the block, the lock is released.

The keyword **synchronized** can also be used as a modifier for methods. For static methods, the *java.lang.Class* object representing the class defining the method is used as lock-object. For instance methods, **this** is used. In both cases, the method body corresponds to the block during whose execution the lock is held.

Other, advanced locking mechanisms are available for Java programs through libraries such as the *java.util.concurrent.locks* API. These mechanisms are, however, out of scope for this paper. The foremost reason for this is the difficulty to associate acquiring and freeing a lock in the source code, which generally correspond to calling methods on a lock object. These method calls can be distributed over different methods and their relation may not be easy to detect. Besides, the **synchronized** keyword is sufficient in most case and therefore commonly used.

In contrast, using the **synchronized** keyword, a lock is only held during the execution of the corresponding statement or method. These locks are mutually exclusive and non-preemptive. Therefore, conditions one and three trivially hold. The second condition (hold and wait) requires that a **synchronized** statement or method must be executed in the control flow of another one. Obvious cases are when **synchronized** statements are nested or a **synchronized** statement occurs inside a **synchronized** method. But there can be less obvious cases as detailed in Section 3.1.

In many cases, we can however assume to encounter one of the more obvious cases of hold-and-wait situations. In the following, we therefore focus on determining whether a circular wait can happen at nested occurrences of **synchronized**.

[3] Java libraries, including the standard API provide additional locking mechanisms, which we exclude in this article, as we explain below.

Our teaching procedure to avoiding deadlocks, therefore consists of six steps, which are further detailed in Section 4:

1. Eliminate alien method calls.
2. Detect possible situations of lock-order deadlock due to a circular wait.
3. Confirm deadlock using a tool.
4. Solve lock-order deadlocks.
5. Check solution using a tool.
6. Reflect on your solution.

3 Ingredients

We intend to explain the six steps in our recommended procedure in more detail. This will be the content of the next section. Before we can do so, we need to do some preliminary work: In section 3.1 we shall characterize several situations in which deadlock can arise, in order to make recognition easier. In section 3.2 we introduce a graphical notation, which we call the Lock-Allocation Diagram (LAD), to help in analyzing such situations. Here we build on results from [3].

3.1 Hold and wait patterns

A deadlock situation where two locks are explicitly requested from the same scope, is relatively easy to detect. Following Dijkstra [4], we shall call this case *Deadly Embrace*. However, in practice similar problems occur that are far less easy to detect.

One complication that may arise is the case where a blocked cycle of more than two threads exist. That is, thread t_1 is holding lock l_1 and waiting for thread t_2 to release lock l_2, thread t_2 is holding lock t_2 and waiting for thread t_3 to release lock t_3, and so on; finally, thread t_n is holding lock l_n and waiting for thread t_1 to release lock l_1. This example was originally posed by Dijkstra as an exam question in 1965 [5]; it has since become known as the *Dining Philosophers*, following its description by Hoare [7]:

> In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. The names of the philosophers were $PHIL_0$, $PHIL_1$, $PHIL_2$, $PHIL_3$, $PHIL_4$, and they were disposed in this order anti-clockwise round the table. To the left of each philosopher there was laid a golden fork, and in the centre stood a large bowl of spaghetti, which was constantly replenished.
>
> A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on his right. When he was finished he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, he just has to wait until the fork is available again.

Finally, there is the possibility that the locks involved are called from different scopes. After acquiring the first lock, a method may call another method, possibly located in a different class, that acquires a second lock. There might even be a dynamically linked sequence of such calls that eventually results in lock acquisition. Whether a scenario leading to deadlock results cannot always be predicted from static code inspections. Hence one often finds the advice [6, page 211] not to call either methods in different classes or methods that may be overridden while a lock is being held. This problem is known as the *Alien Method Call*.

3.2 The Lock-Allocation Diagram

A Lock-Allocation Diagram, or LAD, consists of a number of a Resource-Allocation Graphs (RAG) [9]. It is an extension of the UML activity diagram notation used in part one [3]. An activity diagram is essentially a flow chart, showing the possible flows

of control from activity to activity, and encloses as such a number of possible scenarios of activity executions. A LAD on the other hand, depicts *one* particular scenario of activity executions and thus implies a series of activity states.

In a LAD, we distinguish two types of vertices, namely active threads and objects shared by the active threads. There are also two types of edges, namely edges from a thread to a shared object and edges from a shared object to a thread. A directed edge from a thread to a shared object means that *the thread tries to lock the object* (i.e., execution tries to enter a region protected by the **synchronized** keyword). A directed edge from an object to a thread means that *the object has been locked by the thread* (i.e., execution has successfully entered a region protected by the **synchronized** keyword). The first one is also called a request edge, the second one an assignment edge.

Using the activity diagram notation introduced in part one [3], the deadlock example considered in the introduction (listing 1) can be expressed as shown in the six diagrams in figure 2. In figure 2(a), the first (or outermost) **synchronized** statement is executed: *thread_1* tries to lock object *lock_1*. This request edge is solid and directed from *thread_1* to object *lock_1*.

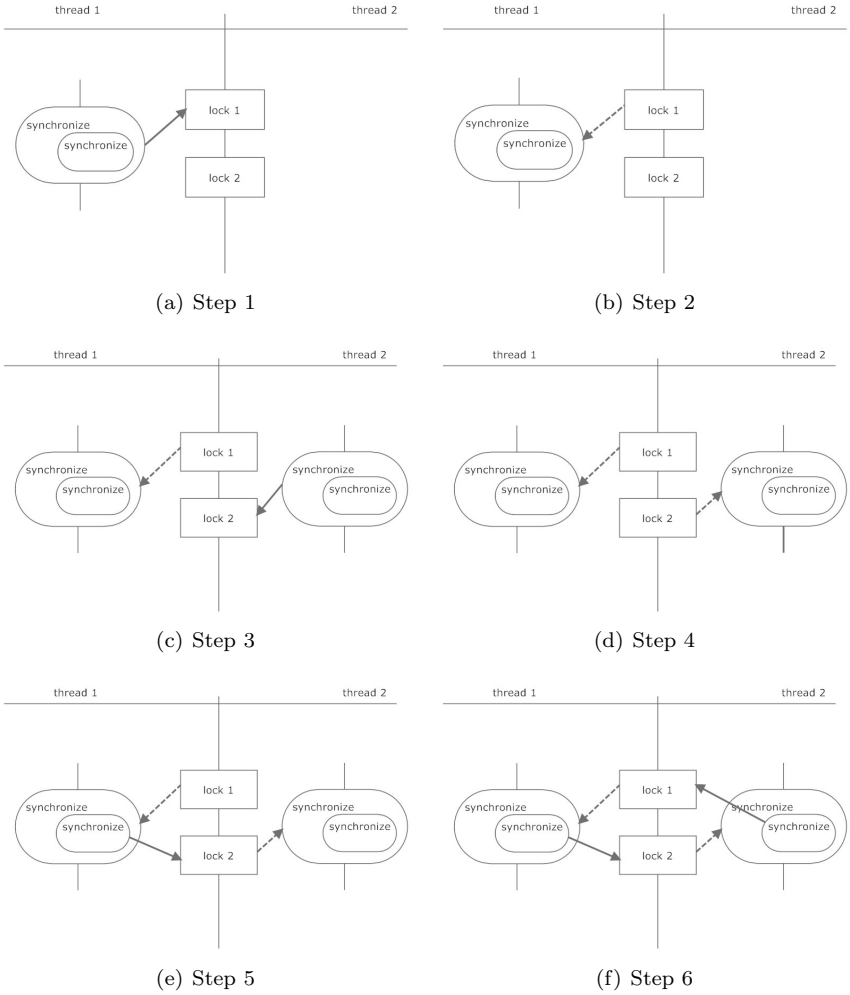
In figure 2(b) the lock has been acquired; the edge has been changed into an assignment edge, i.e., the direction of the edge has been changed and the edge is dotted.

In figures 2(c) and 2(d), the same happens for *thread_2*. The first (or outermost) **synchronized** statement is executed and *thread_2* tries to lock object *lock_2* after which the lock has been assigned.

After that, the second (or innermost) **synchronized** statement of *thread_1* has been executed (figure 2(e)) and *thread_1* tries to lock object *lock_2*. This fails, because object *lock_2* is already locked by *thread_2*. As a consequence, the request edge is not changed into an assignment edge and *thread_2* remains waiting.

Lastly, the second (or innermost) **synchronized** statement of *thread_2* is executed (figure 2(f)) and *thread_2* tries to lock object *lock_1*. This fails too, because *lock_1* is already locked by *thread_1*. Again, the request edge is not changed in an assignment edge and *thread_2* remains waiting, too.

In Figure 2(f) the directed edges form a cycle which mean that a deadlock exists: *thread_1* waits on *thread_2* and *thread_2* waits on *thread_1*.



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6

Figure 2: An example resource allocation graph

4 Procedure in detail

In the following subsections, we will detail the separate steps of our approach.

4.1 Step 1: Eliminate alien method calls

If a method is called while a lock is being held, this introduces the danger of deadlock, because the called method may acquire another lock, and thread interleaving may then lead to blocking situations similar to Deadly Embrace. Even if inspection of the method's code shows that no such lock acquisition takes place, there is no guarantee that overridden versions in new subclasses will never do so. A method in a different class, or an overridable method in the class itself (neither **private** nor **final**) is called an *alien* method, and it is inadvisable to call such an alien method with a lock held. Calling a method with no locks held is known as an *open call*, and open calls are harmless as far as deadlock goes.

We now give a short example to show how the danger might arise in a natural way. Consider a banking system that allows deposits and withdrawals to be performed by many users simultaneously, and also allows for inspection of the account balance at any time. In this case class *Account* will have a **private int** field *balance*, and a **public** method.

```
public synchronized boolean deposit(int amount) {
    int newBalance = balance + amount;
    boolean allowed = (newBalance >= 0);
    if (allowed) balance = newBalance;
    return allowed;
}
```

Next assume that we will also need a method to transfer money from one account to another. This cannot simply be performed by a sequence of calls of *deposit*, as in

```
alice.deposit(−200);
bob.deposit(200);
```

because then inspection of the balances between the two operations would yield an inconsistent state. Hence, in order to make the transfer atomic, we introduce a special-purpose **synchronized** method *transferMoney*. Its code might look like this:

```
public synchronized boolean transferMoney(Account to, int amount) {
    boolean allowed = amount > 0 && deposit(−amount);
    if (allowed) to.deposit(amount);
    return allowed;
}
```

However, this is not an open call, as the call *to.deposit*(*amount*) will acquire a lock on *to* in addition to the lock on **this** already held. Now if one thread executes

alice.transferMoney(*bob*, 200);

and another thread executes

bob.transferMoney(*alice*, 300);

an interleaving like the one in Figure 1 may occur, blocking both threads indefinitely. In the next sections we shall show a deadlock-free way of achieving atomicity of transfers.

Strategies There is no universally applicable procedure for removing alien method calls. One strategy that is often successful is to place the method calls outside the lock-protected critical section [6, page 213]. Another way is to inline all or part of the method bodies. This may create explicitly nested critical sections and a consequent lock-order deadlock situation; but such may be remedied by reordering, as explained in section 4.4 below. If the method calls do not create problems in the present version of the program but may do so in future subclasses, making the method definitions **final** may be all that is required.

4.2 Step 2: Detect possible situations of lock-order deadlock due to a circular wait

A circular wait situation can only occur when a thread already holds a lock and tries to acquire another one. If the code contains no alien method calls, at the beginning of the execution of each method, a thread cannot be holding locks.

Thus, to detect possible lock-order deadlocks, occurrences of nested **synchronized** statements or **synchronized** statements within a method with the **synchronized** modifier must be identified. If no such situations are found, the code does not contain the potential for deadlock and our procedure is finished.

If such situations exist, it must be determined which objects are locked by the respective occurrences of **synchronized**. In particular, it must be determined if potentially the same object is locked by an outer **synchronized** keyword in one scenario and an inner **synchronized** keyword in another scenario. For this purpose, draw the Lock Allocation Diagram to analyze whether a deadlock may occur. Part of this analysis is to identify whether two locked objects may be identical which is generally difficult because of aliasing. If it cannot be excluded that two objects are the same (e.g., because of type incompatibility), it should be conservatively assumed that they may be identical and thus (if the object participates as lock in a circular wait) a deadlock may occur.

For the identified potential deadlock situations, insert *Thread.sleep* calls into the code to enforce the identified deadlocking thread interleaving. Next run the modified program to validate whether a deadlock indeed occurs. If no deadlock occurs, the lock objects identified by means of the LAD are apparently not identical, and we have been over-conservative—the potential for deadlock is thus rejected.

4.3 Step 3: Confirm deadlock using tool

Statical tools only detect deadlock in obvious cases like listing 1. Dynamic tools like JCarder [1] can detect a possible deadlock only if you supply a scenario in which a deadlock can occur. Even if a deadlock has not occurred while running the program, JCarder can indicate that a deadlock might occur. Therefore, you must first write

code that uses a deadlock scenario from step 2. You have to use this code also in step 5 to validate that the deadlock is really solved.

A scenario for booking seats that causes a deadlock is that thread Alice calls *bookSeats* for seat 37 A en 37 B and that thread Charly calls *bookSeats* for seat 37 B and 37 A, see also Figure 3.

4.4 Step 4: Solving lock-order deadlocks

If a possible situation of lock-order deadlock has been detected, requisite measures must be taken.

Deadlock prevention is limited to prevent situations in which circular wait can occur.

There are three possible solutions: firstly use only one lock, secondly in case of multiple locks acquire the locks in a global order and finally use a timed lock.

The simplest form of deadlock prevention is the use of at most one lock; a program that never uses more than one lock at a time cannot get into deadlock [6, page 215]. In many cases, however, this is not a practicable solution: the concurrent program degrates to a sequential program.

In cases of multiple locks, one should try to keep the number of potential locking interactions to a minimum and guarantee that all threads acquire the locks in a global order [6, page 215].

In some cases, the number of locks van be reduced to one (global) lock.

For listing 1 we can replace the two lock objects *lock_1* and *lock_2* with one object *lock* and replace the two synchronized statements with **synchronized**(*lock*). As stated above, this approach leads to a program in which the critical section can only be performed sequentially.

A better solution, however, is to define a global lock-ordering in the program code itself. The program in the introduction (see section 1) is free of deadlock if the locks are acquired in a fixed order, for example first acquire *lock_1* and after that acquire *lock_2*. The following listing shows a modified version of the listing from section 1 wereby the modified lines are highlighted.

```
final Object lock_1 = new Object();
final Object lock_2 = new Object();

Thread t1 = new Thread(new Runnable() {
    public void run() {
        synchronized (lock_1) {
            synchronized (lock_2) {
                System.out.println("do something");
            }
        }
    }
});

Thread t2 = new Thread(new Runnable() {
    public void run() {
        synchronized (lock_1) {
```

```
                synchronized (lock_2) {
                    System.out.println("do something else");
                }
            }
        }
    });

    t1.start();
    t2.start();
```

In cases where the order in arguments passed to a method can cause problems, ordering can be induced by use of identities of objects passed as arguments. Take for example the following implementation of method *transferMoney* [6, pages 207, 208].

```
public void transferMoney(Account from, Account to , int amount) {
    synchronized(from) {
        synchronized(to) {
            from.debit(amount);
            to.credit(amount);
        }
    }
}
```

Deadlock can occur if two threads A and B call method *transferMoney* at the same time with interchanged arguments, for example:

Thread A: *transferMoney*(*alice*, *bob*, 200);

Thread B: *transferMoney*(*bob*, *alice*, 10);

Being out of luck, thread A is holding the lock on *alice* and is waiting for the lock on *bob* and at the same time thread B is waiting for the lock on *alice* and is holding the lock on *bob*. The cause of deadlock is again the nested **synchronized** statements. But the solution, in this case, is to induce an ordering on the locks using the identities of the objects passed as arguments. Suppose each account object has a unique, immutable and comparable account number *number*. Lock-ordering can simply be induced as follows:

```
public void transferMoney(Account from, Account to , int amount) {
    if (from.getNumber < to.getNumber) {
        synchronized(from) {
            synchronized(to) {
                from.debit(amount);
                to.credit(amount);
```

```
                    }
                }
            } else {
            synchronized(to) {
                synchronized(from) {
                    from.debit(amount);
                    to.credit(amount);
                }
            }
        }
    }
```

In cases when argument objects do not have such a key, ordering can be induced by use of Java's *System.indentityHashCode*. Because two objects can have the same hashcode, a third tie breaking lock must be used [6, pages 208, 209]. The third possible solution is to use timed lock attempts using the *tryLock* feature of class Lock. However, we will not use this solution.

4.5 Step 5: Check solution using a tool

After you have changed your code to prevent deadlock, you have to run JCarder and your program again with the scenario from step 3 in order to investigate whether your solution is correct. When JCarder does not indicate a possible deadlock you can be confident that deadlock will not occur for the current scenario. Although, you can not be absolutely sure because JCarder might not be able to find all possible deadlock situations for the given scenarios.

4.6 Step 6: Reflect

Writing test cases for a concurrent program is often harder than writing the concurrent program itself Testing for deadlock in a concurrent program is virtually impossible [6, Chapter 12]. Furthermore, the fact that during testing a deadlock did not occur, does not guarantee the absence of a possible deadlock in your program. As a consequence, to guarantee the quality of your program, a critical evaluation of the decisions made so far is important.

Granularity of synchronization. Walk once more through the steps of the approach and at each step ask yourself whether there are better alternative solutions to consider. It should be mentioned that 'what is better', in the case of concurrency problems, depends on the context. Generally, one can vary the granularity of synchronization by choosing a specific object for locking. In case of a data collection, for example, you can apply a lock on the whole collection, on individual elements, or, as a intermediate form, on a specific group of elements. The first one has the advantage that there is no risk of deadlock at the cost of performance. In both other cases deadlock can occur, but the program's performance is often better.

Documentation. It is strongly advised to document a class's synchronization policy for its maintainers as well as for its clients, for example, what was the synchronization problem and how is it solved?

Code review. It is advised to perform a code review and to use a static tool like FindBugs [6, Chapter 12.4.2]. The code review must also ensure the quality of the documentation.

5 Example

We return to the problem of booking seats for an airplane, treated in detail in a previous report [3, Section 4.2]. As an extension of the original exercise we now consider the case where a passenger wants to book two, possibly adjacent, seats. This should be considered a single transaction, in the sense that no booking is made when one of the desired pair is already occupied.

5.1 First attempt

We expect the main program to contain code like the following:

```
// make booking for Alice for seat 37 A and 37 B
DoubleBooking alice = new DoubleBooking(37,37,'A','B',"Alice",plane);
Thread aliceThread = new Thread(alice);

// make booking for Bob for seat 15 C
Booking bob = new Booking(15,'C',"BOB",plane);
Thread bobThread = new Thread(bob);

// make booking for Charly for seat 37 C and 37 B
DoubleBooking charly = new DoubleBooking(37,37,'C','B',"Charly",plane);
Thread charlyThread = new Thread(charly);
```

The result should be that either Alice books seats 37 A and B and Charly books nothing, or Charly books seats 37 C and B and Alice books nothing. The situation where either of these passengers is left with a single booked seat should be avoided.

As a first attempt, we try to write a class *DoubleBooking* that delegates as much as possible of the work to *Booking*.

```
public class DoubleBooking implements Runnable {
  private Booking booking1, booking2;

  public DoubleBooking(int row1, int row2, char aisle1, char aisle2,
      String name, Plane plane) {
    booking1 = new Booking(row1, aisle1, name, plane);
    booking2 = new Booking(row2, aisle2, name, plane);
  }

  public void tryBooking() throws Exception {
    if (row1 == row2 && aisle1 == aisle2)
      throw new Exception("Attempt by "+name+
        " to book the same seat twice");
    booking1.tryBooking();
    booking2.tryBooking();
```

```
  }

  public void run() {
    try {
      tryBooking();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

However, we soon discover that this does not work, as we get output from this that looks like

```
Try to book for BOB on seat 15 C
Try to book for Charly on seat 37 C
Seat at 37 C booked for Charly
Try to book for Charly on seat 37 B
Seat at 37 B booked for Charly
Try to book for Alice on seat 37 A
Seat at 37 A booked for Alice
Try to book for Alice on seat 37 B
Seat at 37 B not booked for Alice ; already taken by Charly
Seat at 15 C booked for BOB
Seats for this plane:
15 C BOB
37 A Alice
37 B Charly
37 C Charly
```

The problem is that this solution books all seats consecutively and does not protect the transaction character of double bookings. Therefore we should not build *DoubleBooking* from *Booking*, but make it into an intrinsically atomic action.

5.2 Second attempt

In order to combine the booking of two seats, we add a method *bookSeats* to class *Plane* that is completely analogous to *bookSeat*, but takes two seats at a time.

```
public void bookSeats(int row1, int row2, char aisle1, char aisle2,
    String name) {
  seats[row1][aisle1].book(name);
  seats[row2][aisle2].book(name);
}
```

This can be called from *DoubleBooking* as follows:

```
public void book() {
  System.out.println("Try to book for "+name+" on seat "
        +row1+" "+aisle1+" and "+row2+" "+aisle2);
  plane.bookSeats(row1, row2, aisle1, aisle2, name);
}
```

The output now becomes

```
Try to book for BOB on seat 15 C
Seat at 15 C booked for BOB
Try to book for Charly on seat 37 C and 37 B
Seat at 37 C booked for Charly
Seat at 37 B booked for Charly
Try to book for Alice on seat 37 A and 37 B
Seat at 37 A booked for Alice
Seat at 37 B not booked for Alice ; already taken by Charly
Seats for this plane:
15 C BOB
37 A Alice
37 B Charly
37 C Charly
```

So we have not solved the problem. Although the booking of both seats now occurs in a single syntactic unit, viz. the method *bookSeats*, this offers no guarantee of atomicity. Threads belonging to Alice and Charly may execute the individual statements in this method with any interleaving, thus creating a race condition. We need more synchronization to prevent this.

5.3 Third attempt

In order to prevent a passenger requiring two seats from booking only one of them and then finding that the other one is not available, we acquire the lock of both seats before booking. Method *bookSeats* in *Plane* now becomes

```
public void bookSeats(int row1, int row2, char aisle1, char aisle2,
    String name) {
  Seat seat1 = seats[row1][aisle1];
  Seat seat2 = seats[row2][aisle2];
  synchronized (seat1) {
    synchronized (seat2) {
      if (!seat2.isBooked()) seat1.book(name);
      else System.out.println("Seat at "+seat1.getRow()
        +" "+seat1.getAisle()+" not tried for "+name
        +" because second desired seat already taken by "
        +seat2.getPassenger().getname());
```

```
        seat2.book(name);
      }
    }
  }
```

The output now shows a better result:

```
Try to book for BOB on seat 15 C
Seat at 15 C booked for BOB
Try to book for Alice on seat 37 A and 37 B
Seat at 37 A booked for Alice
Seat at 37 B booked for Alice
Try to book for Charly on seat 37 C and 37 B
Seat at 37 C not tried for Charly because second desired seat already taken by Alice
Seat at 37 B not booked for Charly ; already taken by Alice
Seats for this plane:
15 C BOB
37 A Alice
37 B Alice
```

However, our satisfaction is short-lived, because trying the case where Charly books 37 B and A sometimes misfires. Although most of the time correct output like that above is produced, sometimes the program produces only

```
Try to book for Alice on seat 37 A and 37 B
Try to book for Charly on seat 37 B and 37 A
Try to book for BOB on seat 15 C
Seat at 15 C booked for BOB
```

and then stalls.

5.4 Final version

The program's behavior suggests that a deadlock situation may occur. Hence we follow the steps outlined above to find a remedy.

5.4.1 Step 1: Eliminate alien method calls

The **synchronized** section of *bookSeats* contains several calls of methods of *Seat*. At present, these are harmless because, although **synchronized** themselves, they only reacquire the locks on *seat1* and *seat2* already held. This is not a problem because locks in Java are reentrant. Just omitting the **synchronized** from the method declarations is not thread-safe because the method is used elsewhere without synchronization at the point of call, e.g. in method *bookSeat*.

However, this situation still counts as an alien method call, since we cannot be sure that future redefinitions of the methods of *Seat* in subclasses will not acquire other locks, thus introducing new deadlock possibilities. To prevent this, the methods of *Seat* called from within *bookSeats* should be declared as **final**.

5.4.2 Step 2: Detect possible situations of lock-order deadlock due to a circular wait

As before, we may draw a LAD showing the various threads and their actions on the shared *Seat* objects: see Figure 3.

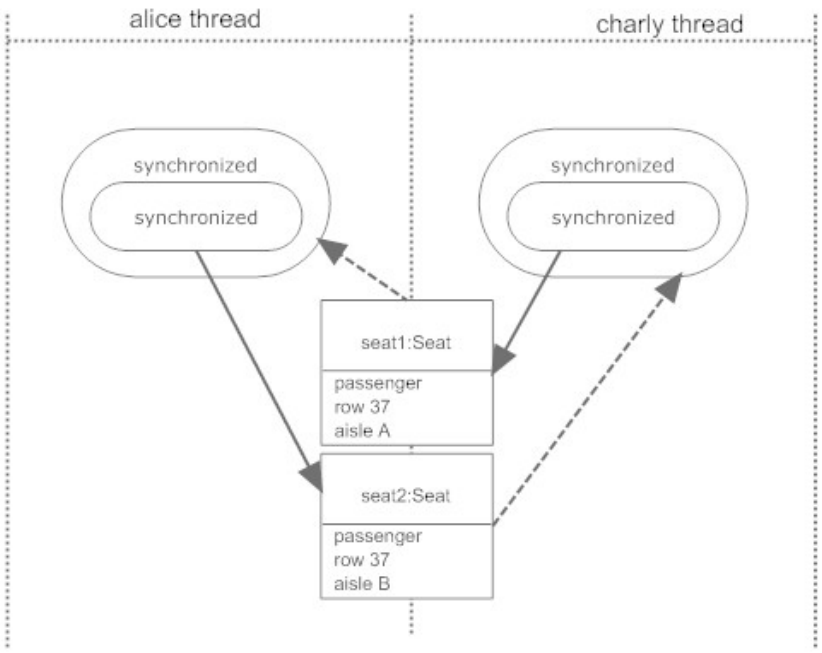


Figure 3: LAD for booking two seats

As can be seen from the LAD, the problem is that the following scenario may take place:

1. Alice acquires the lock on seat 37 A
2. Charly acquires the lock on seat 37 B
3. Alice attempts to acquire the lock on seat 37 B, but finds she must wait until Charly releases it
4. Charly attempts to acquire the lock on seat 37 A, but finds he must wait until Alice releases it

5.4.3 Step 3: Confirm deadlock using a tool

If we run the JCarder tool on this program, its runtime analysis shows up the cyclic dependency through the following output:

```
Cycle analysis result:
   Cycles:          1
   Edges in cycles: 2
```

```
   Nodes in cycles: 2
   Max cycle depth: 2
   Max graph depth: 2
Ignoring 0 gated cycle(s).
Ignoring 0 almost identical cycle(s).
```

This confirms the existence of a cycle liable to cause deadlock.

5.4.4 Step 4: Solving lock-order deadlocks

To prevent such circular waits, the usual remedy is lock ordering. That is to say, in situations where multiple threads require multiple locks, we let them request the locks in a fixed global order. This avoids creating cyclic dependencies. In the current example a natural order on the seats would be the lexicographic order of row and aisle numbers. We therefore introduce a private function

```
private boolean precedes(Seat seat1, Seat seat2) {
  return seat1.getRow() < seat2.getRow()
    || seat1.getRow() == seat2.getRow()
    && seat1.getAisle() < seat2.getAisle();
  }
```

Method *bookSeats* now becomes

```
public void bookSeats(int row1, int row2, char aisle1, char aisle2,
    String name) {
  Seat seat1 = seats[row1][aisle1];
  Seat seat2 = seats[row2][aisle2];
  if (precedes(seat2, seat1)) {
    Seat x = seat1;
    seat1 = seat2;
    seat2 = x;
  }
  synchronized (seat1) {
    synchronized (seat2) {
      if (!seat2.isBooked()) seat1.book(name);
      else System.out.println("Seat at "+seat1.getRow()+" "+seat1.getAisle()+
        " not tried for "+name+
        " because second desired seat already taken by "+
        seat2.getPassenger().getname());
      seat2.book(name);
    }
  }
}
```

This eliminates the danger of deadlock.

5.4.5 Step 5: Check solution using a tool

If we run JCarder once more on the program, the output has changed to

```
Cycle analysis result:
   Cycles:          0
   Edges in cycles: 0
   Nodes in cycles: 0
   Max cycle depth: 0
   Max graph depth: 1
Ignoring 0 gated cycle(s).
No cycles found!
```

The effectiveness of the applied remedy has been confirmed.

5.4.6 Step 6: Reflect

Alternatively, we could have defined *precedes* using a standard facility like *System.identityHashCode* (but be aware that this carries a small risk of hashing collisions).

Rather than using two nested locks, we could also have used a single lock that would apply to *seat1* and *seat2* simultaneously. For instance, the synchronized fragment in *bookSeats* could have been marked as **synchronized**(*seats*). However, this is not satisfactory for two reasons. In the first place, this only precludes interference between simultaneous booking of seat pairs. There is as yet no protection for interference between booking of pairs and single seats; so we would have to change the original method *bookSeat* as well, creating a 'fragile base class' problem. Even more serious is the objection that this would not only prevent simultaneous booking of overlapping seat reservations, but of all reservations, essentially doing away with concurrency and turning the program into a sequential one.

6 Related work and conclusions

6.1 Related work

Several authors have described tools and algorithms for static, and sometimes also dynamic, analysis of possible deadlock occurrences. Ben-Ari [2] describes a tool set for simulating concurrent program execution and for verification by means of model checking. Williams et al. [11] propose an algorithm for static detection of possible deadlock configurations using a lock-order graph. Mehner and Wagner [8] propose an extension to the UML for visualizing mutual exclusion of threads which can be an alternative to the Lock Allocation Diagrams proposed in our work. Their notation also allows to illustrate other concurrency errors such as "dormancy" and thus needs to be more complex than LADs which are sufficient for our procedure. The tool described by von Praun [10] detects several synchronization errors, among them deadlock, and uses both static and dynamic approaches. The static analysis is based on symbolic execution.

Obviously, because of aliasing and non-determinism, none of these tools are capable of giving exact answers to the question whether deadlock will arise; when in doubt, it is preferable that a tool should report false positives to be analyzed by hand.

6.2 Conclusions

In this article, we have presented the first *step-wise plan* to identifying and removing deadlock in software, targeted at the education of first-year students. Our approach is thereby practically oriented: the goal is to deliver working code, not to formally prove its correctness.

This method has already been implemented in new teaching material which will be used in future editions of courses at the participating universities. An in-depth evaluation of the impact on the teaching effectiveness will be the subject of a future report.

Acknowledgement

This project is part of the research group Didactics of Informatics and is carried out in collaboration with E. Barendsen, S. Smetsers, C. Huizing and R. Kuiper. Participating universities in this project are the Radboud University, Eindhoven University of Technology and Open University of the Netherlands.

References

[1] JCarder. `www.jcarder.org`. Accessed: 2015-11-24.

[2] Mordechai Ben-Ari. A suite of tools for teaching concurrency. *SIGCSE Bull.*, 36(3):251–251, June 2004.

[3] A. Bijlsma, H.J.M. Passier, H.J. Pootjes, and S. Smetsers. Didactics for methodical concurrency design. Technical Report TR-OU-INF-2015-01, Open Universiteit, 2015.

[4] Edsger W. Dijkstra. Cooperating sequential processes. `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`, 1965.

[5] Edsger W. Dijkstra. Tentamenopgave cooperating sequential processes. `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD158.html`, 1966.

[6] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley, Upper Saddle River, New Jersey, 2006.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[8] K. Mehner and A. Wagner. Visualizing the synchronization of Java-threads with UML. In *Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on*, pages 199–206, Washington, DC, USA, 2000. IEEE Computer Society.

[9] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[10] Christoph von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2004.

[11] Laurie A. Williams and Robert R. Kessler. All I really need to know about pair programming I learned in kindergarten. *Commun. ACM*, 43(5):108–114, May 2000.