A Structured Design Methodology for Concurrent Programming

Conference Paper · January 2018

DOI: 10.1145/3162087.3162088

Lex Bijlsma
Open Universiteit Nederland
63 PUBLICATIONS **477** CITATIONS
SEE PROFILE

Kees Huizing
Eindhoven University of Technology
6 PUBLICATIONS **333** CITATIONS
SEE PROFILE

Ruurd Kuiper
Eindhoven University of Technology
75 PUBLICATIONS **1,370** CITATIONS
SEE PROFILE

H. Passier
Open Universiteit Nederland
36 PUBLICATIONS **226** CITATIONS
SEE PROFILE

A Structured Design Methodology for Concurrent Programming

A. Bijlsma
Open Universiteit
P.O. Box 9260
6401 DL Heerlen, The Netherlands
lex.bijlsma@ou.nl

C. Huizing
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands
c.huizing@tue.nl

R. Kuiper
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands
r.kuiper@tue.nl

H.J.M. Passier
Open Universiteit
P.O. Box 9260
6401 DL Heerlen, The Netherlands
harrie.passier@ou.nl

H.J. Pootjes
Open Universiteit
P.O. Box 9260
6401 DL Heerlen, The Netherlands
harold.pootjes@ou.nl

J.E.W. Smetsers
Radboud Universiteit
P.O. Box 9102
6500 HC Nijmegen, The Netherlands
s.smetsers@cs.ru.nl

ABSTRACT

Learning how to design and implement a concurrent program is hard. Most textbooks on Java programming only treat concurrency in terms of syntax and examples. They pay little attention to systematically designing concurrent programs. As a result, design experience is to be acquired in a master-apprentice setup of supervised lab classes with immediate, personal feedback.

In this paper, we describe a systematic design method in which the development of a concurrent program is divided into a sequence of explicit, manageable steps which scaffolds students' learning of concurrency concepts and their application. This methodology is intended to improve the procedural development skills of students, providing them with the necessary knowledge and self-efficacy to tackle the problem at hand.

Furthermore, current education moves towards more independent learning. In distance education, for example, immediate feedback on how to proceed in case of problems is often absent. Additionally, enrollment in university courses increased steadily over the last decade which forces educators to spend less time on individual support: students often have to solve problems on their own. To make such independent learning feasible, also as regards providing feedback, a systematic approach is indispensable.

CCS CONCEPTS

•Computing methodologies → Concurrent programming languages;

KEYWORDS

Education, object-oriented programming, concurrency, Java, program design

1 INTRODUCTION

Learning how to program is notoriously hard. Introductory programming courses are generally regarded as rigorous, often having high dropout rates [11, 13]. The situation degenerates further when it comes to concurrent programming. The work reported on here attempts to scaffold students in developing concurrent programs. The method was applied to university students during an object-oriented (OO) programming course (in Java), as part of their first year.

In a concurrent Java program, several threads execute (logically) at the same time. Because these threads will not always run at the same relative speed, the actual order of instructions is non-deterministic. Moreover, threads in Java communicate via shared resources. The combination of non-deterministic execution and sharing gives rise to so-called *race conditions*: the correctness of the program's outcome depends on the timing of the concurrent threads. To remedy this issue, the access to the shared resources must be organized such that threads do not interfere with each other. Therefore, developers of concurrent programs must apply *synchronization* in their code, which means that a thread must acquire an exclusive lock before accessing a critical resource, and release it afterwards.

Difficulties in learning about concurrency. Several studies identify difficulties that students encounter in learning about concurrency. For example, Xie et al. distinguish the following learning problems based on both instructor interviews and student observations [23]:

(1) Instructors use ad-hoc sketches to describe concurrent program executions which students often find difficult to comprehend;
(2) the large number of potential execution sequences makes it difficult to envision and comprehend the dynamic behavior of a concurrent program;
(3) students often mix method execution and scheduling and as such fail to recognize and comprehend race conditions;
(4) students often find it difficult to choose appropriate synchronization mechanisms and primitives, and to reason about why the use of these primitives leads to the expected behavior.

Another issue concerns the correctness of concurrent programs. Students often have several alternative conceptions of correctness, and they usually tolerate errors [10]. Testing, while suitable for sequential programs, is inappropriate for concurrent programs [6].

Furthermore, formal approaches, like model checking and theorem proving, are so complex that they are barely useful in a first-year course [10].

Procedural guidance. To the best of our knowledge, no systematic manner for developing a concurrent program exists which is suitable for an introductory OO programming course. Most textbooks merely introduce the relevant concepts followed by some examples of how they are applied. Consequently, students get stuck on questions like 'How to proceed?' or 'Is my solution correct?'. Therefore, we propose *procedural* scaffolding of students' learning by providing stepwise construction guidelines for concurrent programs. Such an approach can be beneficial in regular educational contexts, where supervision and guidance can gradually be reduced. It is also applicable in distance education, where students typically study the material by self-tuition, individually and mostly at home. Lectures and tutorials are very limited as is the possibility to give (immediate) feedback, being the Achilles heel of distance education [7]. The proposed procedure enables structuring of, possibly partially automated, assessment and feedback.

Research approach. The main objective of our research is to develop procedural guidance throughout our OO-programming course. We follow the educational design research approach [17].

We established, in two cycles, a procedure for the stepwise guidance of students during the development of concurrent programs. Additionally, we investigated the effectiveness of the procedure in our educational context, i.e. 'To which extend does the procedure help students?' and 'Can the procedure be improved?'. The findings so far gave rise to some modifications, resulting in a procedure which is in principle suitable for educational applications.

Contributions. We provide a procedure for developing concurrent programs imparting conceptual knowledge and scaffolding students on how to analyze, design and implement a concurrent program step-by-step. The steps comprise (standard) OO structuring, decisions as to when and how to apply synchronization, reasoning about correctness, and reflection on both implementation and design process. Each step results in a concrete artifact (e.g., a UML diagram or Java code) supporting problem analysis and design decisions. The procedure proposed does not guarantee a 'correct' solution, but supports the students' learning and development processes and in doing so, promotes their independence.

The procedure is intended for an introduction to concurrency in Java as part of a first-year university course in OO programming. We report on the procedure developed and the insights acquired into conditions concerning successful application.

Validation. Teacher reports and exam results have given us reason to believe that providing students with this kind of systematic scaffolding is highly beneficial. In order to extend our understanding of the learning process involved, we have embarked on a more detailed validation program. It is based on recorded observations of students' work and dialogues, as well as in-depth interviews. The results hereof will be addressed in a follow-up paper.

2 PROCEDURAL GUIDANCE

While observing students during think-aloud sessions, we often noticed students falling back on trial-and-error. Once a student literally said: "*I keep on trying as long as needed to get the program working and then ... I take my hands off!*" Here, 'I try' should be understood as *I'm messing around* and 'a working program' is no more than *a program that compiles and produces some output.* These findings are in line with [9]. While students seem to possess sufficient conceptual knowledge to solve program tasks, they are often not able to apply this knowledge effectively and efficiently when designing and implementing a program. What they miss is procedural knowledge which successfully guides them through the steps of completing the task.

From problem solving theory as well as theory of complex learning it is known that conceptual and procedural knowledge are equally important. Moreover, theory of complex learning gives us directions for how to use procedural knowledge in an educational setting.

Problem solving. Software development requires problem solving skills. Soloway [21] discerns five phases in problem solving in case of designing and implementing a program:

(1) Understand the problem;
(2) decompose the problem in order to identify the solution components that will solve the problem components;
(3) select and compose procedures to solve problems, composing the components to a functioning system;
(4) implement plans into language constructs;
(5) reflect and evaluate the final system and the overall design process.

Students experience difficulties mainly in phase one, to set the problem, in phase two, to decompose the problem into parts, and in phase three, to select and compose procedures [18].

Complex learning. According to the theory of complex learning [22], it is essential that students receive supportive information about how to perform a complex task successfully. Supportive information reflects two types of knowledge, namely mental models and cognitive strategies

Mental models: allow students to reason within a domain. This type of knowledge defines what the different concepts are within the task domain (domain model), how these concepts are organized (structural model), and how they work (causal model). For example, a mental model of concurrency is about the concepts threads and shared variables (domain model), which occur in an concurrent application (structural model), and that two threads changing a common variable may cause a race condition (causal model). This type of knowledge is often explained well in textbooks about programming.

Cognitive strategies: allow students to perform a task or to solve a problem systematically. A cognitive strategy specifies the steps to carry out the task, the (overall) order in which the steps should be taken, and the rules (or heuristics) that may be helpful within a step. Cognitive strategies are often missing in textbooks or courses about programming.

Based on these theories, we have developed a procedure for designing concurrent programs which incorporates conceptual knowledge as well as procedural guidance on how to apply this knowledge effectively in order to analyse, design and implement a solution, and to reflect on the result. Initially, students should follow the procedure faithfully. Meanwhile, they are expected to internalize the guidance steps by gradually turning these into procedural knowledge, and after which they apply (details of) the procedure as needed.

3 THE APPROACH

We envisage an introduction to concurrency as part of a first-year university course in OO programming. Students know concepts such as classes, interfaces, and inheritance, and have basic familiarity with UML. The part dedicated to concurrency should take about 20 hours of study. In this time span, we aim to make students acquainted with the following concepts: thread, time-slicing, non-determinism, atomicity, race condition, synchronization, locking and deadlock. We also introduce a minimum amount of Java idiom necessary to use these concepts.

An important consideration of our method is that shared variables accessed from multiple threads must be protected by means of synchronization. The only synchronization mechanism employed here is the protection of methods and blocks of code by annotating them as **`synchronized`**. We propagate a systematic design method from the very start. This enables students to write code themselves from the start and prevents them getting stuck. This way, students get experience in writing concurrent programs of increasing complexity.

The course material accompanying our approach contains several examples illustrating the proposed procedure. Here we mention but one of these, and only briefly: completely worked-out versions of all examples, including source code, may be found in the technical reports [2, 3].

A typical example of a problem students work on is a simulation of a booking process of seats for an airplane. The corresponding assignment starts with a sequential solution which must be converted into a concurrent version. We illustrate the individual steps of our approach by giving example solutions, and discuss the way these are possibly obtained.

We use simple classes, so the focus is on the booking process itself. The plane has 50 rows, each consisting of 6 seats lettered from A to F. Hence, seat 35 A is a valid seat. A passenger is indicated by his name.

3.1 Basics: Threads and Synchronization

Step 1: OO structuring of the problem domain. The students start by analysing the problem in object-oriented manner without reference to concurrency. The resulting design is modeled by a UML class diagram representing the static aspects of the solution. The reason for starting this way is separation of concerns: decisions about the division of responsibilities and the optimal place to store information can be taken without being distracted by concurrency issues. Students can already start with the implementation. As a check, the code should compile.

In the example assignment, the class diagram and a sequential implementation were provided: Students only had to get familiar with both of these.

Figure 1 shows the class diagram of the sequential version of our airplane booking example.

The method `bookSeat` uses method `isEmpty` of class `Seat` to check whether the seat is occupied or not, and if not so, calls method `setPassenger`. In that case a new `Passenger` object will be created and connected to the seat. The boolean return value reflects whether the booking was successful. Figure 2 shows the implementation of `bookSeat`.

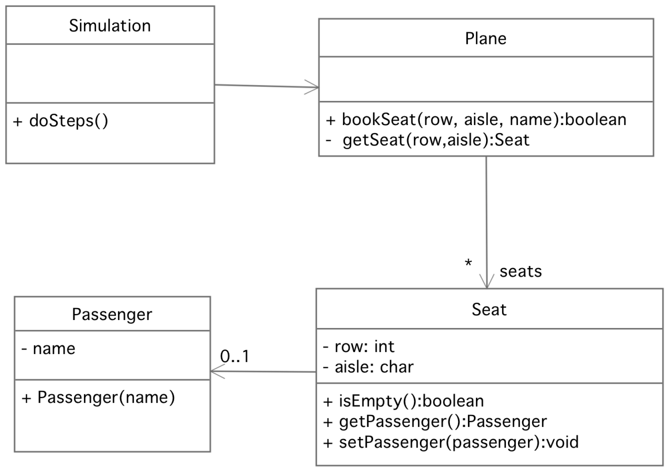

Simulation
+ doSteps()
Plane
+ bookSeat(row, aisle, name):boolean
- getSeat(row,aisle):Seat
*
seats
Passenger
- name
+ Passenger(name)
0..1
Seat
- row: int
- aisle: char
+ isEmpty():boolean
+ getPassenger():Passenger
+ setPassenger(passenger):void

Figure 1: Class diagram of sequential version

```
public boolean bookSeat( int row, char aisle, String name ) {
   Seat seat = seats.getSeat( row, aisle );
   if ( seat.isEmpty() ) {
      seat.setPassenger( new Passenger( name ) );
      return true;
   } else {
      return false;
   }
}
```

Figure 2: Method bookSeat

Step 2: Concurrency of the problem. The students analyze whether concurrency applies, and what type of concurrency is involved. Then they decide which activities should be performed concurrently, which determines the number and roles of threads.

The concurrency design consists of introducing a new class that implements interface `Runnable` for each different role. Each of these so-called *active classes* implements method `run` specifying the activity to be executed in parallel. As run has no arguments and does not return a result, the active class typically contains attributes for holding in- and output values. Moreover, `run` will usually not require real new functionality which means that it can be expressed completely in terms of methods of other classes.

Besides, it has to be decided which class is responsible for managing the active class objects. This class is also likely to create the thread(s).

Then the design can be implemented. The resulting program should compile and run, but no checks on spurious outcomes (e.g., caused by data races) are done yet.

We call the design pattern used in this step the *Active Class Pattern*; see Figure 3 for the general structure with one active class and the application to our example.

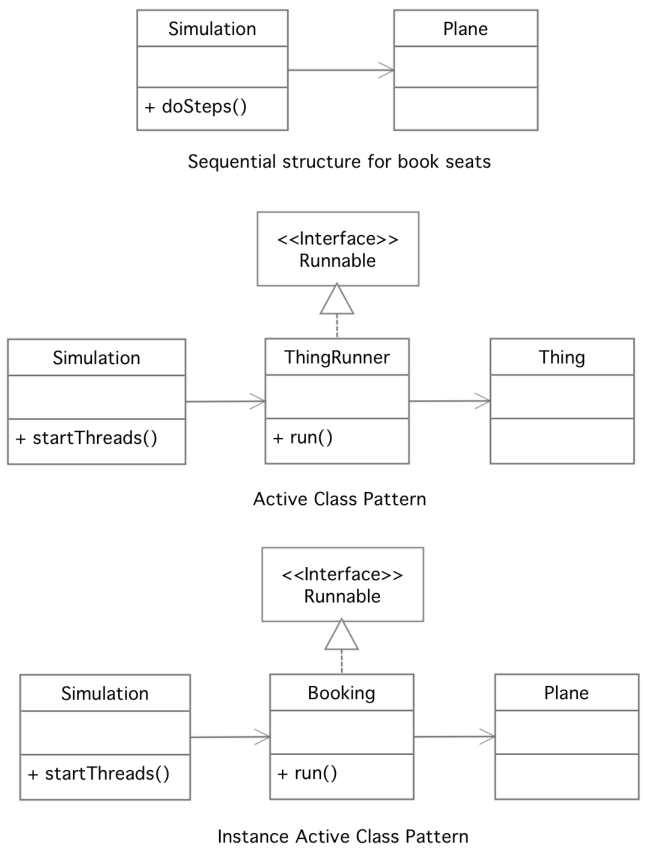


Figure 3: Active Class Pattern

In the example, the process of booking can take place concurrently, and hence it is natural to handle each booking in a separate thread. Therefore, we create a `Runnable` class whose method `run` calls `bookSeat` of `Plane` (see Figure 4).

Step 3: Race conditions (shared variables). At this point, the concurrent program probably is not thread-safe. Typically, it may exhibit faulty behavior due to race conditions involving shared variables. To analyze this, the run time activities are modeled by means of an extended activity diagram, with a separate swimlane for each thread. Creation of threads is shown by means of a fork symbol, and if necessary, waiting for completion of several threads by means of a join symbol. We enhance the standard UML activity diagram by explicitly showing shared objects with attributes accessed from multiple threads, as in Figure 5. The actions in the activity diagram should be expressed in high-level pseudocode, not in detailed program code. In order to handle race conditions, the following rule suffices: for any mutable variable that may be accessed by more than one thread, all accesses to that variable must be performed with the same lock held. Because of the introductory character of the course, we limit ourselves to Java's basic synchronization idioms: synchronized methods and synchronized statements. In the first case the object on which the method is being invoked is automatically used as a lock. In the second case the programmer can, in principle, choose any object. Obviously, this is only safe when the chosen lock object is used consistently. The synchronization strategy of Java is also known as the *Java Monitor Pattern*.

```
public class Booking implements Runnable {
   private int    row;
   private char   aisle;
   private String name;
   private Plane  plane;

   public Booking( int row, char aisle, String name, Plane
        plane ) {
      this.row   = row;
      this.aisle = aisle;
      this.name  = name;
      this.plane = plane;
   }

   @Override
   public void run() {
      ...
      boolean success = plane.bookSeat( row, aisle, name );
      if ( success ) {
         ...
      }
   }
}
```

Figure 4: Class booking

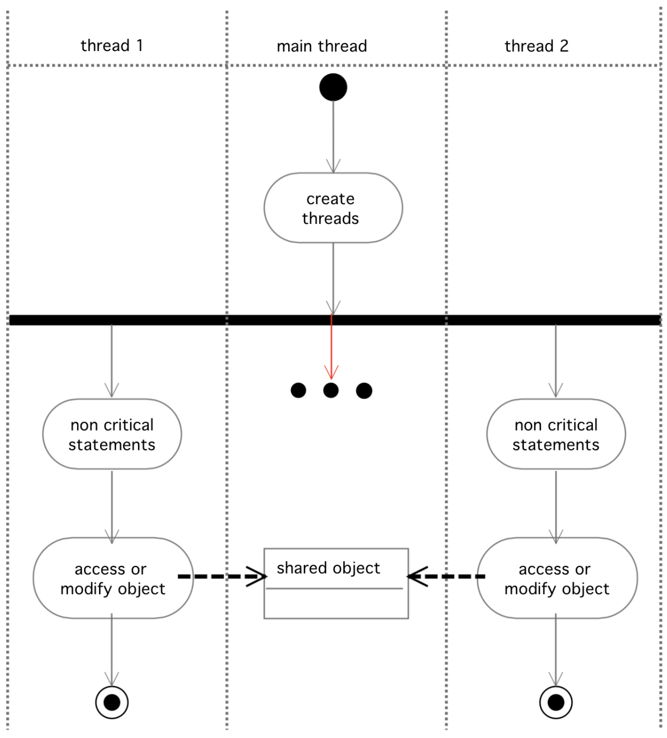


Figure 5: Enhanced activity diagram

A somewhat more sophisticated requirement applies when several variables that are linked through a global invariant. In that case, their updates must occur within the same synchronized method.

This will ensure that no inconsistent intermediate states become visible to other threads.

This step is complete when all shared variable access is regulated via synchronization. As we shall see in Section 3.2, the application of nested locks can cause problems. However, at this stage we assume that this situation does not occur.

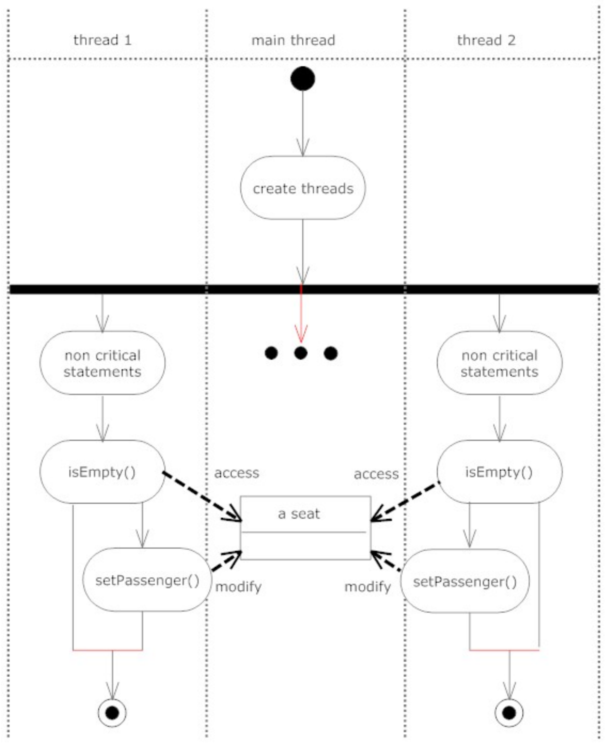


Figure 6: Enhanced activity diagram for booking example

When we draw the activity diagram for the seat booking example, we find that all threads access the common attribute `seats` of `Plane`. This attribute contains a collection of seat objects which is created and filled with empty seats in the constructor of class `Plane`. During execution this collection will never be modified. Seat reservations are administered by updating the corresponding seat objects. Therefore, these seat objects are drawn as shared objects in the activity diagram (Figure 6). Consequently, to protect the `Seat` objects, we declare all methods in the `Seat` class as **`synchronized`**.

Step 4: Race conditions (check-then-act). A special case of race conditions is *check-then-act*: a situation where threads check the value of a common variable and, based on the result, they try to change state of the variable. By the time a thread performs the update, the value of the variable may have been changed by some other thread. These situations should be identified by code inspection. To remedy these problems, the actions of checking and acting must occur in the same synchronized method.

This ensures that the state on which a decision is based is the same as the state on which the update is effectuated.

In our example, there is indeed such a danger because the result returned by `seat.isEmpty()` may already have been invalidated by

```
public boolean bookSeat( int row, char aisle, String name ) {
   Seat seat = seats.getSeat( row, aisle ) ;
   synchronized( seat ) {
      if ( seat.isEmpty() ) {
         seat.setPassenger( new Passenger( name ) );
         return true;
      } else {
         return false;
      }
   }
}
```

Figure 7: Method bookSeat improved

the time `seat.setPaasenger( ... )` is invoked. Making both methods **`synchronized`** does not protect us from this danger. Instead, the test and the booking should be combined into one synchronized block. The code of `bookSeat` now becomes as showed in Figure 7.

In this example, after the seat has been booked, it will not become free again. So it does not make sense to wait for it becoming available. In general, these conditions may change and it does make sense to wait and try again later. To deal with such issues in Java, programmers often use Java's *wait* and *notify* mechanism. This leads to a strategy called the *Producer-Consumer Pattern*. The treatment of this pattern is not part of our basic procedure. It is planned to be included in the part 'advanced topics' of the procedure (see Section 7).

Step 5: Reflection. The nondeterministic nature of concurrent executions makes it is hard to debug or test a concurrent program. A concurrency-related bug may only show up under certain rarely occurring thread interleavings. As a consequence, to guarantee the quality of a program, a critical evaluation of the decisions made is important. Did we synchronize on the right object? Could a greater degree of concurrency have been reached by choosing smaller synchronized sections? It is advisable to keep the synchronized methods as small as possible such that other waiting threads will not be blocked for longer than is strictly necessary. Students can even elaborate on the question of whether classes are *thread-safe* (i.e., do not impose additional synchronization demands when used in a concurrent context). Therefore the students are expected to reflect on the design decisions made and go back to the appropriate step if remedy is required.

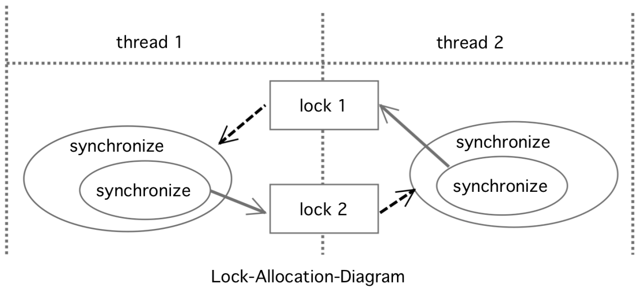
Up to this point we have covered the more straightforward issues concerning concurrency. Deadlocks, which is an even more advanced problem area, is treated in the next section.

3.2 Advanced: deadlocks

Concurrently executing threads that want to acquire a lock already in use, e.g., by invoking a synchronized method or entering a synchronized statement, must wait until it has been released. As a consequence, excessive or careless use of locking can result in deadlocks: two or more threads, already holding a lock, are all waiting for one of the others to release its lock, a so-called *circular wait.*

In this section, we describe how students can recognize situations where a deadlock could arise and how to take appropriate measures to prevent them. However, as opposed to the race-condition issues that were amenable to analysis and could be solved by applying their corresponding remedies, deadlock is too complex to be treated in the same manner. Instead, a *development rule* is proposed. The rule is motivated by a general analysis of deadlock caused by circular waits. This instruction would preferably be part of lectures rather than being included in our procedure. In order to explain our approach, we now provide this analysis as an intermezzo.

Intermezzo: Lock-order deadlock. A circular wait can only occur when a thread already holds a lock and tries to acquire another one. We analyze the situation as follows. In order to detect potential deadlocks, nested synchronized blocks (i.e., synchronized statements or synchronized method calls occurring in a synchronized section) should be identified first. If such situations exist, it must be determined which objects are potentially used as locks for synchronization. In particular, it must be determined if threads can be scheduled in such a way that the same object is locked by concurrently executing an outer and inner `synchronized` statement. To visualize such a scenario, we introduce *Lock Allocation Diagrams*, see, e.g., Figure 8. A Lock Allocation Diagram, or LAD, is



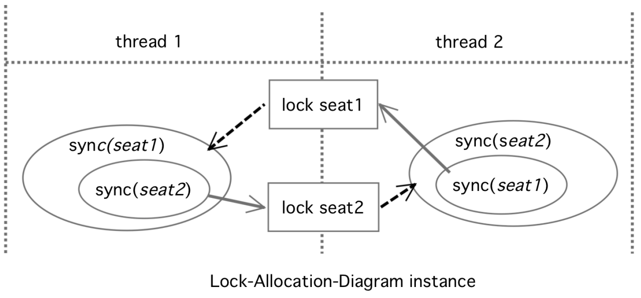


Figure 8: Lock-Allocation-Diagram

a Resource-Allocation Graph (RAG) [20], in which processes are equipped with synchronization statements. We depict a LAD using the enhanced activity diagram introduced in the third step (see Figure 5).

In a LAD, we distinguish two types of vertices: threads and objects shared by these threads. There are also two types of (directed) edges, namely edges from a thread to a shared object and vice versa.

An edge from a thread to a shared object indicates that the thread attempts to acquire the object's lock (i.e., tries to enter a region protected by that lock). An edge from an object to a thread indicates that the object has been locked by the thread (i.e., it has successfully entered the protected region). The first one is also called a *request edge*, the second one an *assignment edge*.

```
public boolean bookSeats( int row1, char aisle1, String name1,
                          int row2, char aisle2, String name2 ){
   Seat seat1 = seats.getSeat( row1, aisle1 );
   Seat seat2 = seats.getSeat( row2, aisle2 );
   synchronized ( seat1 ) {
      synchronized ( seat2 ) {
         if ( seat1.isEmpty() && seat2.isEmpty() ) {
            seat1.setPassenger( new Passenger( name1 ) );
            seat2.setPassenger( new Passenger( name2 ) );
            return true;
         } else {
            return false;
         }
      }
   }
}
```

Figure 9: Method bookSeat with nested locking

The goal of the analysis is to invent an plausible scenario leading to a circular wait. This corresponds to a LAD containing a cycle. Before attempting to construct such a scenario it can be helpful to insert calls of `Thread.sleep` into the code at appropriate places to enforce the possible deadlock, and, subsequently, to run the program one or more times. This may already prove that a deadlock is really possible, and hence a deadlock scenario must exist. If a possible situation of lock-order deadlock has been detected, requisite design measures must be taken.

The simplest form of deadlock prevention is the use of at most one lock: a program where every thread never acquires more than one lock at a time cannot get into deadlock [6, page 215]. This is not always a practicable solution, however: the concurrent program may degenerate into a sequential one. It should be checked whether this is the case.

End of intermezzo.

Step 6: Nested locks. We now consider programs where nested locking is natural or even necessary, i.e., we now go beyond the assumption, stated in step three above, that nested locking should not occur.

We extend the original example by considering the case in which a passenger wants to book two, possibly adjacent, seats. This method `bookSeats` should be considered as a single action. No booking is to be made if one of the desired pair of seats is already occupied. We lock both seats prior to booking. This prevents a passenger requiring two seats from booking only one of them and then finding out that the other one is not available any more. The method `bookSeats` in `Plane` may look something like in Figure 9.

Step 7: Lock-ordering deadlock. Without any precautions, this straightforward implementation has the danger of deadlock due to circular waits. This can be shown using the analysis method outlined above, which will result in a DAG similar to the one given in Figure 8. Rather than using two nested locks, we could have used a single lock that would apply to both seats simultaneously. This is not satisfactory, however. This would not only prevent simultaneous booking of overlapping seat reservations, but of *all* simultaneous reservations. In essence, this alternative is equivalent to an implementation in which all `Plane` methods are synchronized, which we already rejected because it would lead to a solution that is essentially sequential.

```
public int compareTo (Seat seat) {
   if ( getRow() == seat.getRow() ){
      return getAisle() - seat.getAisle();
   } else {
      return getRow() - seat.getRow();
   }
}
```

Figure 10: Method compareTo

```
public boolean bookSeats( int row1, char aisle1, String name1,
                          int row2, char aisle2, String name2 ){
   Seat seat1 = seats.getSeat( row1, aisle1 );
   Seat seat2 = seats.getSeat( row2, aisle2 );
   if ( seat1.compareTo ( seat2 ) < 0 ){
      synchronized ( seat1 ) {
         synchronized ( seat2 ) {
            return tryToBook( seat1, name1, seat2, name2 );
         }
      }
   } else {
      synchronized ( seat2 ) {
         synchronized ( seat1 ) {
            return tryToBook( seat1, name1, seat2, name2 );
         }
      }
   }
}

private boolean tryToBook( Seat seat1, String name1,
                           Seat seat2, String name2 ){
    if ( seat1.isEmpty() && seat2.isEmpty() ) {
       seat1.setPassenger( new Passenger( name1 ) );
       seat2.setPassenger( new Passenger( name2 ) );
       return true;
    } else {
       return false;
    }
}
```

Figure 11: Method bookSeats deadlock free

If from the analysis it follows that multiple locks are unavoidable, the best locking policy is to acquire locks in a globally fixed order. If the locks are all from the same class, this can easily be achieved by requiring that this class implements the `Comparable` interface. Nested synchronization should then obey the order indicated by `compareTo`.

In the case of the airplane booking example, we should let class `Seat` implement `Comparable<Seat>` by defining a `compareTo` method straightforwardly, using the numbering and lettering of the seats, as showed in Figure 10. Method `bookSeats` now becomes as shown in Figure 11, where the seats, using method `comparableTo`, are locked in a fixed order.

This eliminates the danger of deadlock.

In cases when multiple lock classes are involved one could let all these classes implement `Comparable`, and use the lexicographic ordering based on the names of the classes in combination with `compareTo`.

If it is impossible to adjust lock classes, one could use Java's `identityHashCode`. Because two objects can have the same hashcode, a third *tie breaking lock* is required [6, pages 208,209].

Once students get more experienced, it comes natural that certain steps in our procedure are combined, for example, step 3 and 4, and step 5 and 6. These are a clear instances where scaffolding (the two separate steps) is gradually replaced by craftsmanship.

Table 1 summarizes the steps of the complete proposed design procedure.

4 FIRST EXPERIENCES

The procedure outlined in the previous section, except the steps dealing with deadlock, has been incorporated into educational material used throughout courses of various levels at three different universities. We focus here on the experiences in distance education at the Open University of the Netherlands (OUNL), as explicit procedures are expected to be most important in a situation where the students are isolated and have less opportunity for learning through imitation [14].

At OUNL, the material was used first in 2015 in a blended-learning setting that involved, next to printed and online texts, a small number of face-to-face discussions. This took place in a first course in OO programming in Java. Some UML diagrams had already been introduced and used earlier in the course. Almost all participants had some professional experience in IT development. At the time, the procedure was slightly different from the steps outlined above; the original version is described in our technical report [3]. Based on an analysis of the students' conceptual difficulties, the original advice to let `Runnable` be implemented by one of the existing classes, was dropped: Often this required too many changes of existing code. Instead, the Active Class Pattern was introduced. Moreover, more attention was payed to the detection of check-then-act situations.

The second run of the material, in which these improvements were incoporated, took place in 2016. According to the instructor's report, the students did surprisingly well in their assignments, leading to high marks. They seemed to follow the procedural steps faithfully. In order to obtain a better understanding of how these steps functioned in scaffolding their thinking, two think-aloud sessions of 3 hours each were recorded and transcribed.

During the think-aloud sessions, the students were indeed observed to refer back to the procedural steps and patterns presented in the teaching material. They performed the steps in the order suggested. However, an unexpected difficulty surfaced: in the exercise, a sequential version of the program was given that simulated concurrency through a step function imitating the active objects stepwise in a round-robin fashion. It turned out that this simulation was confusing (or at least not explained enough beforehand), as the students did not consider discarding the sequential simulation code while introducing active classes and creating the threads.

At the other two regular universities using the same material (Eindhoven University of Technology, Radboud University Nijmegen), similar improvements were less noticeable. Possible explanations for this difference are:

- OUNL students, being used to courses with little or no face-to-face contact, are more accustomed to explicit procedural guidance;

Step	*Activity*	*Results*
1	OO structuring of the problem domain without concurrency	1) Class diagram modeling the static aspects, 2) Implementation of the sequential model
2	Analyzing concurrency of the problem	1) Extended class diagram including active classes, 2) Tentative implementation of the concurrent model
3	Analyzing race conditions (shared variables)	1) Enhanced activity diagram, 2) Revised implementation with synchronized methods for shared variable access
4	Analyzing race conditions (check-then-act)	1) Adapted enhanced activity diagram, 2) Revised implementation with atomic checking and acting
5	Reflection	Results of reconsidered steps
6	Applying nested locks	Tentative implementation with nested locks
7	Analyzing and eliminating deadlock	1) Lock allocation diagram, 2) Final implementation with locks in a globally fixed order

Table 1: The procedure summarized

- all the students entering the OUNL Software Engineering master program have professional experience in IT or related disciplines, and usually have had previous exposure to (company-enforced) explicit methodologies;
- other OUNL courses strongly emphasize (UML-based) modeling, as opposed to the more code-oriented approach at the other universities.

We aim to elicit more insight into the range of applicability of the procedure in a subsequent evaluation study. In particular, it is an open question whether previous exposure to textscuml is essential for the procedure to be successful.

5 RELATED WORK

We are not aware of a procedure describing the development of concurrent programs stepwise, suitable for a first-year course about OO programming. Most standard books about Java describe the syntax elements followed by some examples. Books dedicated to concurrency, as for example [6], describe design rules, patterns, and mental models to support reasoning about concurrency, but lack the procedures of how to use these rules and models effectively in relationship.

A kind of exception is [12] in which a modeling approach is described focusing on dynamical aspects. The approach starts with the specification of requirements. After that, it dives into the modeling of dynamical aspects, i.e. the main actions and interactions are identified and grouped together in components, which are then structured into an architecture. Analysis of safety and liveness is done by translating these aspects into properties which are subsequently verified using model checking tools. The approach is well-structured and rigorous, but too sophisticated as part of an introductory course on OO programming.

Fekete [4] proposes a methodology for teaching undergraduates the design of thread-safe classes, leaving more general synchronization issues aside. Interestingly, the advice is comparable to our 'Fifth Step' in Subsection 3.1, but no use is made of UML or any other aids to visualisation. The overview of frequently made mistakes is very useful.

Mehner and Wagner [15] use UML sequence diagrams to identify deadlock situations. In order to analyze the causes of deadlock they introduce a number of extensions of UML collaboration diagrams, for example time constraints and special-purpose stereotypes. The formalism is quite expressive, but also has a tendency to become complicated, and in order to reduce the complexity, the authors propose a way to decrease the number of active objects. Schader and Korthaus [19] analyze the possibilities of modeling concurrency with standard UML. They prefer activity diagrams (as we do); the difference is that they leave the synchronized objects implicit but make the locking itself explicit by means of split and join operations.

In *How to Design Programs* [5], the authors introduce a systematic design method for functional computer programs using the concept of a design recipe: a six-step process for creating programs from the problem statement. Similar to our approach, each step of the design process scaffolds a student by graphical notations and rules and results in a concrete artifact. In [16] this idea is exploited in the context of concurrent programming in the programming language Racket. Their design recipe for distributed programming consists of seven steps, starting from a division of the problem at hand into components, and ending with a testing procedure. The major difference with our approach is the underlying model of concurrency: Whereas we use threads and thread communication via shared memory, the distributed processes in [16] are not created explicitly, and communicate via message passing.

Finally, several authors have reported on tools and algorithms for static as well as for dynamic analysis of possible deadlock situations; e.g., see [1]. Obviously, because of aliasing and non-determinism, none of these tools are capable of giving exact answers to the question whether deadlock is possible. It is, moreover, questionable whether these tools can be successfully applied in a first-year course. The time it takes to master such tools, often outweighs the advantages of their use.

6 DISCUSSION

We conjecture that our method alleviates the learning difficulties mentioned in the introduction. Applying a structured approach during the design of concurrent programs gives students more grip on the complex task at hand. The proposed procedure scaffolds students in designing and implementing a concurrent program as confirmed by our observations and preliminary analysis of experimental results.

The use of ad-hoc sketches is replaced by a precise methodology based on UML, preventing diagrams whose meaning suffers from ambiguity. In the first step, construction of a UML standard class diagram forces the students to analyze the problem domain in an OO manner, without being distracted by concurrency issues. In the second step, the Active Class Pattern is applied and added to the UML class diagram. In the third step, we employ a variant of the UML activity diagram to envision and comprehend the dynamic behavior of the concurrent program. The activity diagrams used here are standard UML diagrams enhanced with the facility to highlight object attributes referenced from more than one thread. These departures from the standard are motivated by the goal to aid in analyzing possible runtime conflicts. If done with sufficient care, the use of such diagrams will guard against typical student mistakes such as synchronizing write operations, while leaving read operations unsynchronized. In our approach the use of UML is restricted to class and activity diagrams. In principle one could replace UML by any other formalism provided that there is consensus about the meaning and it is applied consistently.

Reasoning about the effects of concurrent execution is facilitated by the stepwise organization of our procedure admitting the analysis of different problem aspects in isolation. The reflection step at the end serves to review the solution and determine whether the objectives of introducing concurrency into the problem have indeed been attained. The structured approach might also rectify student's conception of correctness [8]: one cannot conclude that a concurrent program behaves correctly just by executing it a few times. Explicit checks performed after each step as well as a reflection at the end of the procedure are indispensable.

7 CONCLUSIONS AND FUTURE WORK

The first results of our approach for designing concurrent programs are promising. For the next run, we will re-examine each step of our procedure and, when necessary, add more detail and/or examples (with special attention to the problem of how to transform a sequential simulation into a concurrent one). We will also cover the *Producer-Consumer Pattern* at a high level. Race conditions are now handled solely by using synchronized methods and synchronized statements. Our plan is to employ more sophisticated techniques, as, for instance, offered by Java's concurrency framework (`java.util.concurrent` and `java.util.atomic`) in order to solve race conditions and other concurrency issues.

Our long-term research goal is to develop comprehensive procedural guidance, consisting of rules, notations, and accompanying steps that aid students to analyse and solve problem tasks. We are striving for the conversion of our study material from mainly syntax driven to a more problem driven approach. Other subjects that we plan to incorporate are the *Model-View-Controller Pattern*, event handling, and unit test design. Generally, the students should be able to classify a certain problem and our educational material should guide them explicitly by providing a sequence of logical steps that lead to a solution. These steps are supported by (graphical) analysis methods and dedicated design patterns.

REFERENCES

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

[2] A. Bijlsma, C. Bockisch, H.J.M. Passier, and H.J. Pootjes. Methodical concurrency design in education, part two: Deadlock. Technical Report TR-OU-INF-2015-01b, Open Universiteit, 2016.

[3] A. Bijlsma, H.J.M. Passier, H.J. Pootjes, and S. Smetsers. Methodical concurrency design in education, part one: Race conditions. Technical Report TR-OU-INF-2015-01a, Open Universiteit, 2015.

[4] Alan D. Fekete. Teaching students to develop thread-safe Java classes. *SIGCSE Bull.*, 40(3):119–123, June 2008.

[5] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Krishnamurthi Shriram. *How To Design Programs, An Introduction to Programming and Computing.* The MIT press, Cambridge, Massachusetts, Londen, England, 2001.

[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] G. Guri-Rosenblit. Distance Education and E-Learning: Not the Same Thing. *Higher Education*, 49 (4):467–493, 2005.

[8] Yifat Ben-David Kolikant. Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies*, 60(2):243 – 268, 2004.

[9] Yifat Ben-David Kolikant. Students' alternative standards for correctness. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 37–43, New York, NY, USA, 2005. ACM.

[10] Jan Lönnberg, Anders Berglund, and Lauri Malmi. How students develop concurrent programs. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 129–138, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[11] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*, 39(1):499–503, 2007.

[12] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs 2nd Edition.* Wiley, 2006.

[13] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulletin*, 33(4):125–180, 2001.

[14] Tanya McGill and Valerie Hobbs. A supplementary package for distance education students studying introductory programming. In *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '96, pages 73–77, New York, NY, USA, 1996. ACM.

[15] 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.

[16] Marco T. Morazán. Functional video games in CS1 III - distributed programming for beginners. In *Trends in Functional Programming - 14th International Symposium, TFP 2013, Provo, UT, USA, May 14-16, 2013, Revised Selected Papers*, volume 8322 of *Lecture Notes in Computer Science*, pages 149–167. Springer, 2014.

[17] T. Plomp. Educational design research: An introduction. In T. Plomp and N. Nieveen, editors, *Educational design research*, pages 10–51. Enschede: SLO, 2013.

[18] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[19] M. Schader and A. Korthaus. Modeling Java threads in UML. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language*, pages 122–143. Physica-Verlag, 1998.

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

[21] E. Soloway, J. Spohrer, and D. Littman. E unum pluribus: Generating alternative designs. In *Teaching and Learning Computer Programming*, pages 137–152. Lawrence Erlbaum Associates, 1988.

[22] Jeroen J.G. van Merriënboer and Paul A. Kirschner. *Ten Steps to Complex Learning, a systematic appraoch to four-component instructional design.* Taylor & Francis, New York, NY, USA, second edition, 2013.

[23] Shaohua Xie, Eileen Kraemer, and R. E. K. Stirewalt. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 727–731, Washington, DC, USA, 2007. IEEE Computer Society.