Methodical Concurrency Design in Education Part one: Race conditions Technical Report (TR-OU-INF-2015-01a)

A. Bijlsma[*1], H.J.M. Passier[†1], H.J. Pootjes[‡1] and J.E.W. Smetsers[§2]

[1] *Open Universiteit, Faculty of Management, Science and Technology, Postbus 2960, 6401 DL Heerlen, The Netherlands*
[2] *Radboud Universiteit, Institute for computing and information Sciences, Postbus 9102, 6500 HC Nijmegen, The Netherlands*

December 17, 2015

[*] lex.bijlsma@ou.nl
[†] harrie.passier@ou.nl
[‡] harold.pootjes@ou.nl
[§] s.smetsers@cs.ru.nl

Abstract

Learning how to design and program software is hard. Learning how to design and program a concurrent program is even much harder. Most text books about Java programming treat threads and concurrency in terms of syntax and examples, but often don't pay much attention to how to design such a program. In this report, we describe a systematic design method in which the development of a concurrent program is divided into six manageable steps. An important aspect of our method is the use of (extended) UML as a visualization tool. The method follows the proven dialectical approach 4C/ID, which is dedicated for learning complex tasks as designing and programming concurrent programs is. To show the method at work, we describe two examples in detail.

1 Introduction

Learning how to program is notoriously hard. Novice programmers seem to suffer from a wide range of difficulties and deficits. Introductory programming courses are generally regarded as rigorous, often having high dropout rates. The situation degenerates further when it comes to concurrent programming.

OO programmers have learned to consider an object-oriented system as a collection of collaborating objects which communicate with each other in order to reach a specific goal. In this light, each object can be seen as an autonomously operating entity. Equivalently it can be seen as a specialized concurrent process. Unfortunately thread-objects, in Java for instance, do not correspond with this perspective. To understand the essence of threads, knowledge about the underlying execution model of OO programs is indispensable. Technicalities, such as call-stack and sequential flow of control, which could be avoided in the original OO view, are suddenly required in order to understand the role of thread objects in an OO software system. This makes concurrent programming more complex than OO programming.

In addition to this mandatory change of view, concurrent programming itself is also a complex task. In particular, the unpredictability of interactions between concurrent executing threads, makes it hard to take all possible execution scenarios into account while developing a concurrent program. Moreover, the communication and synchronization primitives, for example in an OO language like Java, that are needed to protect shared data from being corrupted, appear to be much more complex than the concepts which are used to build sequential systems.

Several studies have been carried out in order to identify the difficulties that students encounter in learning about concurrency. In [13], for example, research based on both instructor interviews and student observations distinguishes the following learning problems:

1. Instructors use ad-hoc sketches to describe concurrent program executions. Students often find it difficult to comprehend the details and explanations, either for later review or for the analysis of new execution behavior.

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. As such they fail to consider execution sequences in which a thread is interrupted before it has finished the method call, while another thread is scheduled which executes the same method on the same object.

4. Students often find it difficult to choose appropriate synchronization mechanisms and primitives to meet certain synchronization goals. Additionally, they have trouble reasoning about why the use of these primitives leads to the expected behavior.

Concurrent programming is perceived as a sophisticated job. There is a need to break the solution down into smaller, more manageable tasks, while maintaining a bird's-eye view of the task at hand.

In this paper we propose a systematic design method in which the development of a concurrent program is divided into a series of manageable steps. An important aspect of our approach is the use of UML as a visualization tool, similarly as is done in OO programming. The first step is to analyze whether the use of concurrency is advisable for the problem at hand. Next, we identify classes which are then modelled in a class diagram. Then, candidate classes for performing concurrent processes are determined. These so-called active classes are implemented as threads. The run-time behavior of these threads can now be depicted by means of an enhanced UML activity diagram. This diagram also includes the (passive) objects shared by multiple threads. The following step is to protect shared objects against simultaneous access by using thread synchronization. Communication diagrams are used to analyze security and lifeness properties, such as the absence of deadlock or starvation. Finally, we consider the possiblity of making classes thread-safe to facilitate reuse.

We conjecture that our method avoids the previously mentioned learning difficulties. Applying a structured approach in dealing with concurrent programming could give the students more grip on the complex task at hand. Using a standard class diagram as a starting point forces the students to specify an initial solution to a problem in an object-oriented manner, without being distracted by concurrency issues. The next phase is meant to help students identify which classes are candidates for being executed in a parallel fashion. Instead of ad-hoc sketches, we use commonplace UML activity diagrams to visualize concurrent program execution. Determining whether synchronization is needed and which objects should be synchronized is done in the subequent step. The activity diagrams in addition to the communication diagrams help students reason about currency in a more systematic manner, and thereby increasing comprehension.

2 Didactic approach

Programming is a complex task. This is especially true for designing and implementing concurrent programs, because these programs are intrinsically complex. The didactic approach 4C/ID is dedicated for learning complex tasks [10, 11]. This approach consists of series of learnings tasks, in which supportive information, procedural information and practising are integrated. Complex learning involves the integration of knowledge and skills.

An environment for complex learning should have the following four components [10, 11]:

Task description This is the task itself and should be as authentic as possible. In our case, the task is to solve a problem where concurrency is involved.

Supportive information and guidance This type of information is needed for learning and performing problem-solving in the domain of interest. It explains how the domain is organized, e.g. which concepts are important and how these concepts are interrelated, and how problems should generally be approached. Supportive information, with a focus on the products involved, describes the initial state of a problem, the goals that should be reached, and the solutions that get a student from the initial state to the goals. Guidance, with a focus on solution processes, describes the schema-based processes needed for successfully solving a task. A schema describes the solving processes on a higher level. It can be performed in a variable way depending on the characteristics of the problem faced. Supportive information and guidance are crucial for performing non-routine aspects that often involve problem solving and reasoning; they help the student in determining 'the next step', which concepts play a role and how should the sub-problem faced be approached.

Procedural information Procedural information has a focus on solution processes on a more lower level. A procedure specifies how to perform a routine aspect and takes the form of a precise, step-by-step, description. It is performed in a highly consistent way from problem situation to problem situation.

Part-task practice For those aspects classified as routine, additional part-task practice may be provided. Such practises are exercises that help students to reach a higher level of automation.

In de rest of this paper, we focus on the supportive and procedural information for developing simple concurrent programs.

3 Teaching objectives

We envisage a first introduction to concurrency as part of a first-year university course in object-oriented programming. The part dedicated to concurrency should take about 20 hours of study.

In this time span, we aim to familiarize the students with the following concepts: thread, call stack and control flow, time-slicing, scheduler, context switch, nondeterminism, atomicity, race condition, synchronization, critical section, locking. We also show a minimum amount of Java syntax necessary to use these concepts.

We emphasize common reasons to introduce concurrency into a program, such as improvement of efficiency and processor utilization, avoidance of non-responsiveness, and simulation of inherently parallel processes in the real world.

The design principle followed in our method is that mutable state variables accessed from multiple threads must be protected by means of synchronization. The only means of synchronization employed here are the protection of object methods and blocks of code therein by locks. In order to enable the students to work on realistic problems, we employ a systematic design method from the very start, even with problems that are easily solved informally or where much of the work has been done beforehand. We now discuss the steps in this design method.

3.1 First step: problem analysis

The students start by analyzing the problem in order to establish whether it is amenable to the use of threads. As stated above, reasons for using threads may refer to efficiency, responsiveness or simulation.

3.2 Second step: class design

The students will first design the classes for an object-oriented solution to the problem without any reference to concurrency. This design is modeled by an UML class diagram. 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 the complications concurrency introduces.

3.3 Third step: selecting active classes

The introduction of concurrency is done by choosing which relevant class or classes from the previous design should implement interface `Runnable`. This means that those classes should implement a method `run()` that contains the tasks to be executed in parallel. The class selected should be the one that contains the data necessary to perform these tasks, as follows from OO design principles. Sometimes it suffices to rename one of the existing methods to `run()`, but as this method may take no arguments, deliver no result and raise no

exceptions, usually some amount of refactoring is necessary. Another approach is to call one or more of the existing methods from run. Sometimes it is necessary to introduce a new class. It is advisable not to call any methods of Swing components from `run()`.

3.4 Fourth step: modeling threads

If `C` is an active class as defined in the previous subsection, and `c` an instance of `C`, a thread can now be created and started by means of `new Thread(c).start()`. The runtime activities can now be modeled by means of an UML activity diagram, with a separate swimlane for each thread. Creation of threads is shown by means of a fork symbol, waiting for completion of several threads by means of a join symbol. We enhance the standard UML activity diagram by explicitly showing objects accessed from multiple threads, as in Figure 1. The actions in the activity diagram should be high-level pseudocode, not detailed program code.

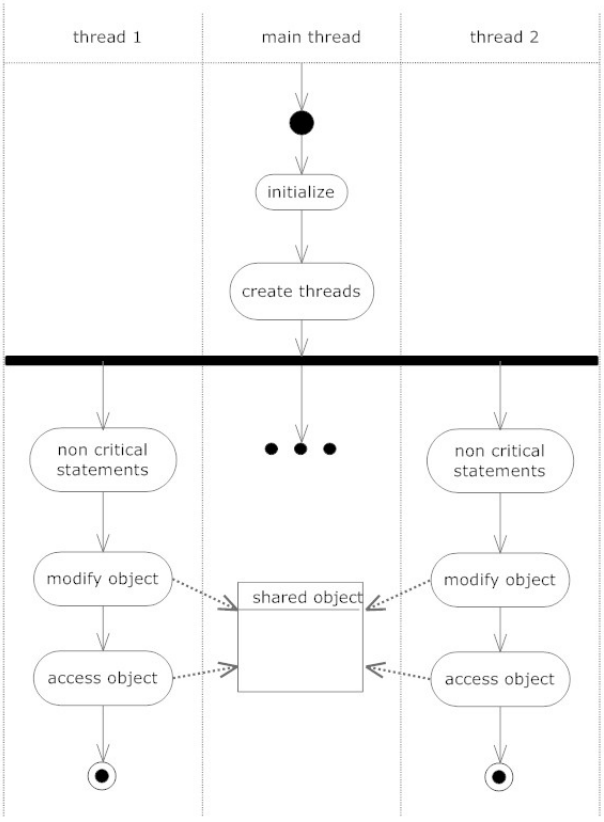

Figure 1: Enhanced activity diagram

3.5 Fifth step: Synchronization

At this point, the concurrent program shown in the activity diagram probably is not thread-safe. Typically, it will exhibit nondeterministic behavior due to race conditions. A race condition occurs when the result of executing the program is dependent on the relative timing of different threads. This is the case when a thread inspects a variable that may have been modified by a different thread.

In order to prevent 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 synchronized methods, which means that the entire method body is guarded by a single lock, namely the object on which the method is being invoked. This strategy is known as the *Java monitor pattern*.

In particular, if correctness of the code depends on the condition that the value of an inspected variable has not been changed since it was set in the current thread, the actions of setting and inspecting must occur in the same synchronized method. This ensures that decisions based on the inspected value reflect its current state. If several variables are linked through a global invariant, their updates must occur within the same synchronized method. The latter rule ensures that no inconsistent intermediate states become visible to other threads.

Apart from this, it is advisable to keep the synchronized methods as small as possible, by eliminating statements that need not be synchronized and by avoiding long calculations within the method body. In that way other threads will not be blocked for longer than is really necessary.

3.6 Sixth step: Reflection

After the program has been completed and tested, there remain some questions to be answered. We may ask whether a greater degree of concurrency could have been reached by choosing smaller synchronized sections, and whether the classes are thread-safe, i.e. can be used in other programs without imposing synchronization demands on their clients.

3.7 Does the method fit the didactic approach?

As we have seen, the didactic approach requires, as far as the parts on which we focus in this paper, supportive guidance as well as procedural information.

Supportive information describes how the domain is organized in terms of concepts and how these are interrelated. The teaching method starts with the introduction of all relevant concepts, for example: thread, call stack and control flow, and atomicity. Using this domain description, students are already able to reason about small concurrency problems without using Java syntax. Because we use Java as programming language, the basic thread concepts as class `Thread` and interface `Runnable` are introduced. The *initial state* contains the problem description and the constraints that should be satisfied. The initial state of a

typical concurrency problem in a first-year university course consists of a short case description. The constraints that should be satisfied are besides thread-safety and general design principles, a founded design, which is an important step in the schema. The *goal state* is formed by the products as result of each of the steps of the method, i.e. a UML class diagram, classes marked as active, an UML activity diagram that models the run time activities of threads, a description of the measures taken (here applying the Java monitor pattern) and a reflection on the process followed and the product delivered in the end.

Because designing and programming concurrent programs is a non routine activity, the *guidance information* consists of a schema consisting of six steps. Each of these steps is on a high level and describes non procedural activities as designing a class diagram and modeling the run time behavior of threads using the activity diagram notation. As we have noticed, each of these steps results in a product underpinning the quality of the final product. Furthermore, the steps help students in filling in the gap between an exercise description and the final solution.

4 Examples

This section presents two examples. The first example is a small one in which a number of players concurrently throw one shared dice. In this assignment, an initial non thread-safe Java program is the starting point. We ask a student to analyze the program using the schema and make the program thread-safe. The second example, the reservation of airplane seats, is a more complex one.

4.1 N players, one dice

A number of players throw randomly one shared dice. The implementation of class `Dice` is as follows:

```
public class Dice {

  private int pips = 1;
  private Random rand = new Random();
  public static final int MAX = 6;

  public void throwTheDice() {
    pips = rand.nextInt(MAX) + 1;
  }

  public int getPips() {
    return pips;
  }
}
```

All players are playing with one and the same dice. The implementation of class `Player` is:

```
public class Player implements Runnable {

  private Dice dice = null;
  private String name = null;
  private int numberOfThrows;

  public Player(String name, Dice dice, int numberOfThrows) {
    this.name = name;
    this.dice = dice;
    this.numberOfThrows = numberOfThrows;
```

```
  }

  public void run() {
    throwTheDice();
  }

  private void throwTheDice() {
    for (int i = 0; i < numberOfThrows; i++) {
      dice.throwTheDice();
      int pips = dice.getPips();
      System.out.println(this.name + " throws " + pips);
    }
  }
}
```

Now, we let two players play with one dice:

```
public class LetsPlay {
  public static final int PLAYERS = 2;

  public static void main(String[ ] args) {
    dice dice = new Dice();
    for (int i = 0; i < PLAYERS; i++) {
      Player p = new Player("p" + i, dice, 100);
      Thread t = new Thread(p);
      t.start();
    }
  }
}
```

Assignment. This program is not thread-safe. Analyze the program following the six steps and improve it. Try to visualize, using console output, the lack of thread-safeness.

4.1.1 First, second and third step: analysis, class design and selecting active classes

In this case, the use of threads is already given; the program falls into the simulation category. The division of responsibilities can be determined from the implementation given. The resulting class diagram is shown in figure 2. Because the threads are already implemented, the active class (`Player`) is marked as active during this step as well.

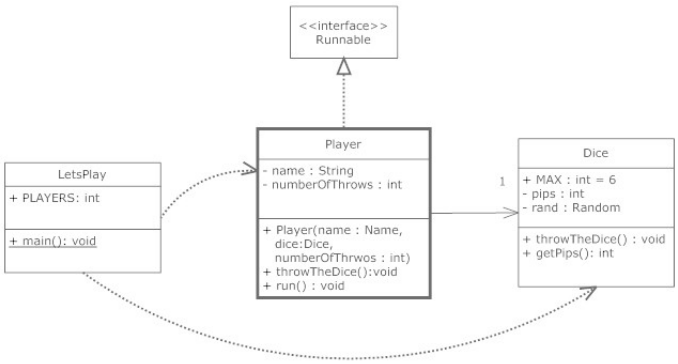

Figure 2: Class diagram with Player as active class

4.1.2 Fourth step: modeling threads

The runtime activities are now modeled using a UML activity diagram, see figure 3, assuming that two players throw the shared dice. It will be clear that this

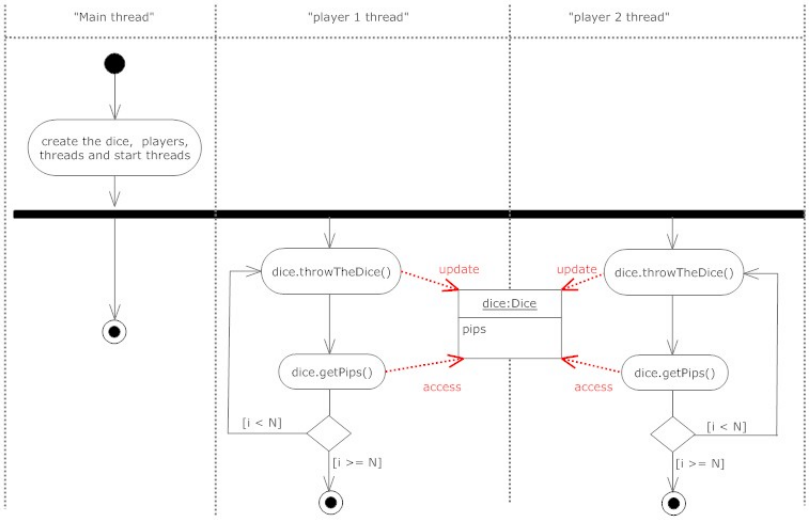

Figure 3: Activity diagram for two players and one shared dice

program is not thread-safe. If a context switch occurs during throwTheDice, it is possible that the number that will be stored in pips is not the actual number that has been thrown by the dice. So throwTheDice must be synchronized. However synchronizing throwThedice is not sufficient, because if a context switch occurs between the compound method calls `dice.throwTheDice()` (write) and `dice.getPips()` (read) the value of attribute pips changes by a concurrent thread. As a result, the number of pips thrown does not equal the number of

pips read. We can easily show this situation by changing the implementations of method `throwTheDice()` of class `Player` and method `throwTheDice()` of class `Dice`:

```
// Changed method throwTheDice in class Dice
public int throwTheDice() {
  pips = rand.nextInt(MAX) + 1;
  return pips;
}

// Changed method throwTheDice in class Player
private void throwTheDice() {
  for (int i = 0 ; i < numberOfThrows ; i++) {
    int throwed = dice.throwTheDice();
    int read = dice.getPips();
    if (throwed != read) {
      System.out.println(throwed + " - " + read);
    }
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {}
  }
}
```

An example output is:

```
p0 throws: 2, reads: 5
p1 throws: 3, reads: 5
```

4.1.3 Fifth step: Synchronization

To avoid race conditions we have to design a compound synchronized method that throws the dice and returns the number of pips. The code of this method, throwAndReadThePips, reads:

```
public synchronized int throwAndReadThePips() {
  throwTheDice();
  return pips;
}
```

4.1.4 Sixth step: reflection

As a variation on the solution, one can give method `throwTheDice` in class `Dice` a return value of type `int` representing the number of pips thrown. A disadvantage of this solution is that the interface of class `Dice` changes.

4.2 Booking seats for an airplane

In this exercise we will simulate the process of booking seats for an airplane. We start with a sequential solution which must be changed to a concurrent version in the assignment. We will use simple classes, so the focus is on the booking process itself. The plane has 50 rows of seats placed in aisles from A to F. So seat 35 A is a valid seat. A passenger is indicated by his name. Figure 4 shows the class diagram of the sequential version. The code of method main of class

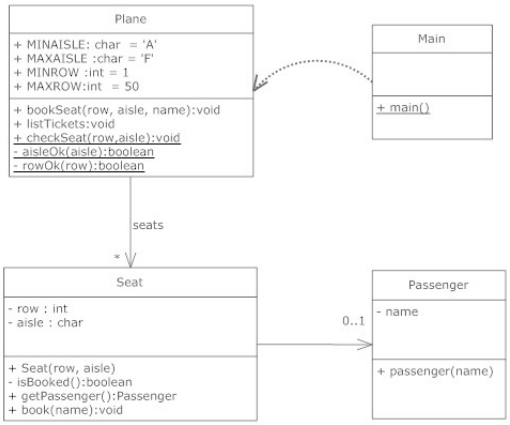

Figure 4: Class diagram of sequential version

`Main` reads:

```
public class Main {

  public static void main(String[ ] args) {
    Plane plane = new Plane();
    try {
      // book seat for Alice at 37 A
      plane.bookSeat(37, 'A', "Alice");

      // book seat for Bob at 15 C
      plane.bookSeat(15, 'C', "Bob");

      // book seat for Charly at 37 A (which is already occupied)
      plane.bookSeat(37, 'A', "Charly");
    } catch (Exception e) {
      System.out.println(e);
    }
    System.out.println("Seats for this plane:");
    plane.listTickets();
  }
```

```
}
```

The output reads:

```
Seat at 37 A booked for Alice
Seat at 15 C booked for Bob
Seat at 37 A not booked for Charly ; already taken by Alice
Seats for this plane:
15 C Bob
37 A Alice
```

Method `bookSeat(row, aisle, name)` of class `Plane` checks whether the combination of row and aisle is valid, and if valid, calls method book(name) of class `Seat`. This method checks whether this place has not been booked yet. In that case a new `Passenger` will be created and connected to the seat and a message of success will be printed. If the seat has already been booked, only a message of failure will be printed. The implementation of bookSeat and `checkSeat` is as follows:

```
/**
  * Books a seat for a passenger
  * @param row row number
  * @param aisle aisle character
  * @param name name of the passenger
  * @throws Exception if row, aisle is invalid
  */
public void bookSeat(int row, char aisle, String name)
              throws Exception{
  checkSeat(row, aisle);
  seats[row][aisle].book(name);
}

/**
  * Check whether the combination of row and aisle is valid
  * @param row row number
  * @param aisle aisle character
  * @throws Exception
  */
public static void checkSeat(int row, char aisle) throws Exception {
  if (!Plane.rowOk(row)) {
    throw new Exception("Row " + row + " is not valid");
  } else {
    if (!Plane.aisleOk(aisle)) {
      throw new Exception("Aisle " + aisle + " is not valid");
    }
```

```
  }
}
```

The complete code of all the classes is available in project BookingSequential (see appendix).

Assignment. Use the steps of the design method to design and implement a concurrent version. Take the classes of the sequential version as the basis of your version.

4.2.1 First and second step: analysis and class design

The process of booking can take place concurrently. So it is obvious to handle each booking in a separate thread. This is again an example of the simulation category.

The classes being already given, there is no work to be done in the second step. The first question that we have to answer is which class will implement interface `Runnable`.

4.2.2 Third step: selecting active classes

If we look at the class diagram 4, we see two possible classes to implement interface Runnable: `Plane` or `Main` but both solutions are unsatisfactory. This is an example of the situation where we have to introduce a new class which will implement interface `Runnable`. We call this class `Booking` and give it attributes `row`, `aisle`, `name` and `plane`. In this way, a booking object can have a method `run` which will call `bookSeat` of `Plane`. We then get the class diagram of figure 5. In method `main` we initialize a Plane object and start threads with a Booking object as an argument.

4.2.3 Fourth step: modeling threads

The corresponding activity diagram is shown in figure 3. We have taken the same bookings as in the sequential version. All threads access and possibly modify the common attribute `seats` of `Plane`. So we can expect problems if we don't synchronize. In order to have the seats properly printed at the end of the program, we have to join the threads. A possible output of the unsynchronized version reads:

```
Seat at 15 C booked for BOB
Seat at 37 A booked for Alice
Seat at 37 A booked for Charly
Seats for this plane:
15 C BOB
37 A Charly
```

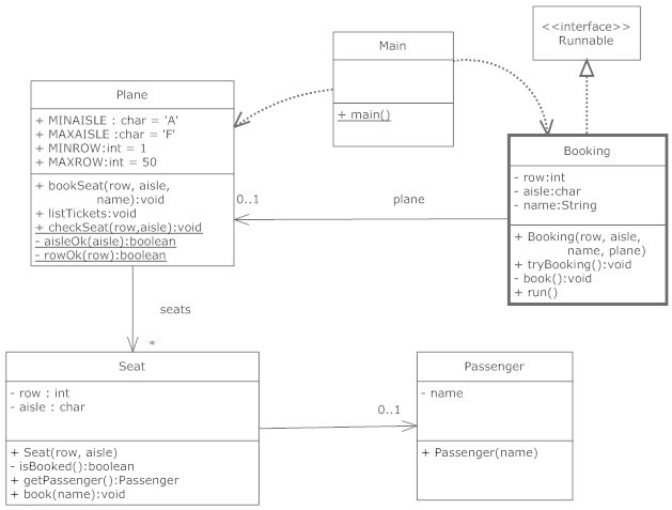

Figure 5: Class diagram of concurrent version

It is clear that this output has an error: seat 37 A was booked for Alice but has been overridden with Charly.

4.2.4 Fifth step: synchronization

We can solve the problem by making method bookSeat of class Plane synchronized:

```
/**
  * Books a seat for a passenger
  * @param row row number
  * @param aisle aisle character
  * (Precondition: row, aisle are valid)
  * @param name name of the passenger
  */
public synchronized void bookSeat(int row, char aisle, String name) {
  seats[row][aisle].book(name);
}
```

4.2.5 Sixth step: reflection

If we analyze figure 6 a little better, we can reason that the synchronisation problem only occurs when two or more thread access and modify the same seat, not the complete array of seats. So we create a new diagram which is depicted

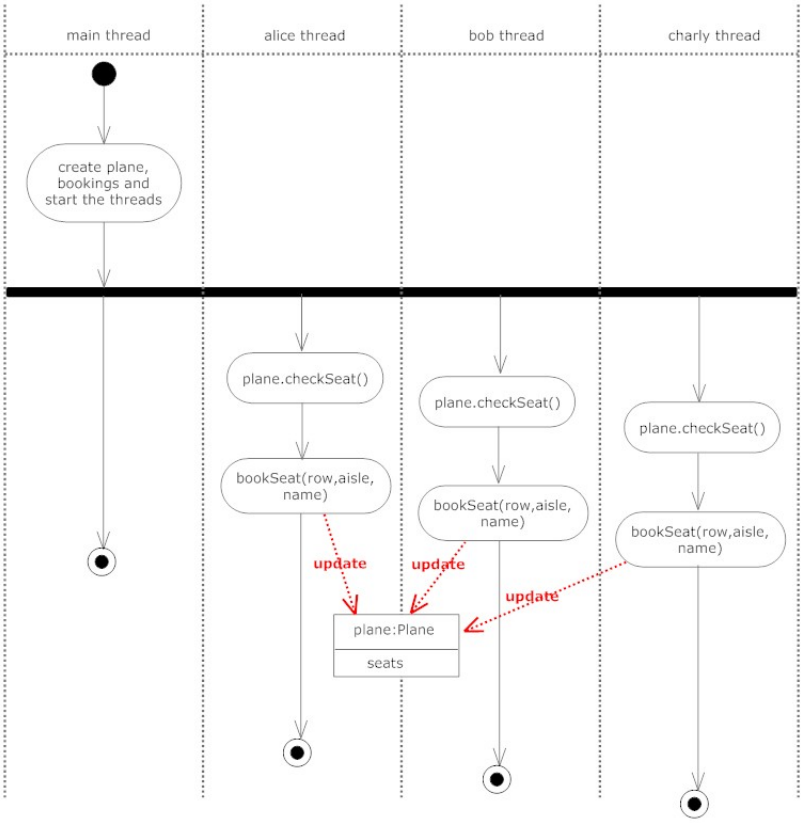

Figure 6: Activity diagram of concurrent version

in figure7. We also have taken in account the join of the threads in order to print out the list of seats. Method `book` of class `Seat` now reads:

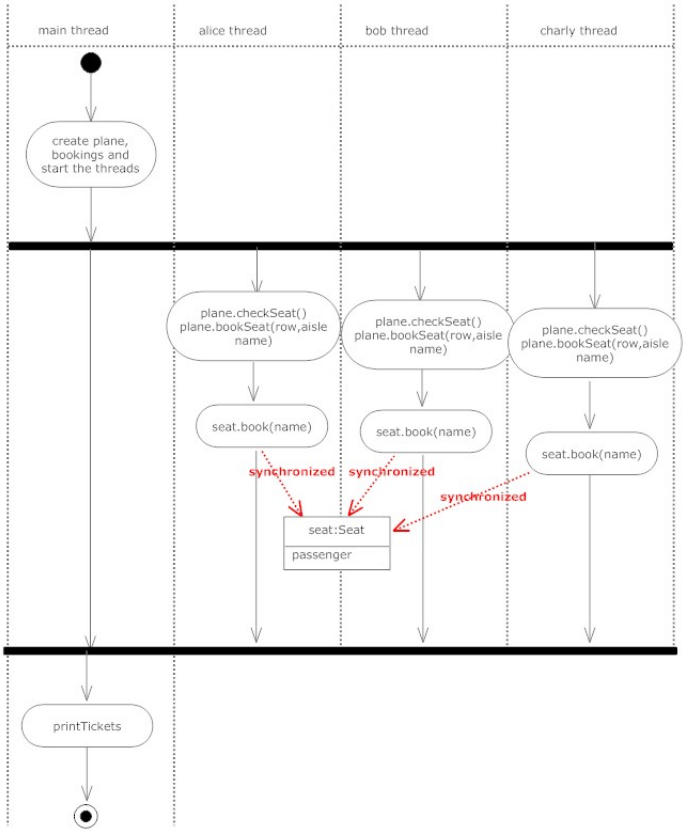

Figure 7: Optimized activity diagram of concurrent version

```
/**
  * Book this seat
  * @param name the name of the passenger
  *
  */
public synchronized void book(String name) {
  if (!isBooked()) {
    this.passenger = new Passenger(name);
    System.out.println("Seat at " + row + " " + aisle + " booked for "
                                    + passenger.getname());
  } else {
    System.out.println("Seat at " + row + " " + aisle
```

```
                                + " not booked for "
                                + name + " ; already taken by "
                                + passenger.getname());
    }
}
```

5 Related work

Fekete [1] proposes a methodology for teaching undergraduates the design of thread-safe classes, leaving more general synchronization issues aside. The advice is virtually identical to the 'Fifth Step' in subsection 3.5, but no use is made of UML or any other aids to visualisation. The overview of frequently made mistakes is very useful.

Goñi and Eterovic [2] use OCL for specifying the semantics of concurrency primitives, but do not offer a design methodology.

Mehner and Wagner [3] 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, e.g. time constraints and special-purpose stereotypes. These have a tendency to become very complicated, and the authors also propose a way to reduce the number of active objects.

Schader and Korthaus [6] 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.

Stevens [9] also analyzes the possibilities in standard UML 1.4, but does not propose extensions.

Xie et al. [13] describe how interviewing students and teachers enabled them to pinpoint conceptual difficulties in an existing concurrency course. Their remedy is to visualize thread behavior by means of UML sequence diagrams extended with locking and state information.

6 Future work

6.1 Validation

As stated in the introduction, we *conjecture* that our method avoids learning difficulties students experience, i.e. we expect that applying this structured approach in dealing with concurrent programming gives students more grip on the complex task of designing and implementing a concurrency program. To test our expectations, we are validating the approach in an one trimester, first year course about Java programming at three universities, i.e. two regular universities (Radboud University Nijmegen and Eindhoven University of Technology) and one distance university (Open University of the Netherlands). At the regular universities, the lectures are given in a (physical) lecture-room. At the Open University, the lecture is given using the electronic classroom Blackboard Collaborate[1].

For teaching the approach, we are organizing one lecture of two hours followed by a tutorial of two hours in which the students will make three exercises of increasing complexity. At each university, about 15 students will be selected. Each lecture is supported by detailed handouts, containing theory, the approach, examples and small exercises. Next to the lecture and tutorial, the selected students will take a written exam consisting of a selected exercise.

All the lectures will be sound-recorded. After the lectures, tutorials and exams are finished, a number of students will be interviewed at each university. These interviews will consist of an open as well as a closed part.

As part of the tutorials, two students are selected at each university. Each of these couples are solving an extra exercise using pair-programming [12]. The solving processes will be video- as well as sound-recorded. Because pair-programming requires intensive mutual consultations, we will have detailed information of the cognitive processes at our disposal, comparable to data obtained by a think-aloud session [8]. Each of these pair-programming sessions will be finished by a video-stimulated recall interview [4], in which the student couples view video sequences of their process followed and are invited to reflect on their decision-making processes.

6.2 Extension of the approach

Our approach so far considers only the protection of mutable state variables accessed from multiple threads by means of synchronization, i.e. application of the Java monitor pattern. In the next future, we will extend the approach by adding application of the Producer-consumer pattern and as part of that analyzing and preventing of deadlock situations.

[1] See http://www.blackboard.com/Platforms/Collaborate/Overview.aspx

Acknowledgement

This project is part of the research group Didactics of Informatics and is carried out in collaboration with E. Barendsen, 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] Alan D. Fekete. Teaching students to develop thread-safe Java classes. *SIGCSE Bull.*, 40(3):119–123, June 2008.

[2] Agustín Goñi and Yadran Eterovic. Building precise UML constructs to model concurrency using OCL. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004: The Unified Modeling Language. Modeling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 212–225. Springer, Berlin Heidelberg, 2004.

[3] 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.

[4] Nga Thanh Nguyen, Amanda McFadden, Donna Tangen, and Denise Beutel. Video-Stimulated Recall Interviews in Qualitative Research. In *AARE 2013 Conference Proceedings*, 2013.

[5] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[6] 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.

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

[8] M. van Someren, Y. van Barnard, and J. Sandberg. *The think aloud method: A practical guide to modelling cognitive processes*. Academic Press, Londen, 1994.

[9] Perdita Stevens. UML and concurrency. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 151–166. Springer, Berlin Heidelberg, 2003.

[10] Jeroen J.G. van Merriënboer, Richard E. Clark, and Marcel B.M. De Croock. Blueprints for complex learning: The 4c/id-model. *Educational Technology Research and Development*, 50(2):39–61, 2002.

[11] 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.

[12] 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.

[13] 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.