Integrating a Simulation Case Study into CS2: Developing Design, Empirical and Analysis Skills

Kay A. Robbins, Catherine Sauls Key and Keith Dickinson
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
{ krobbins,key, kdickins}@cs.utsa.edu

Abstract

Case studies are widely used in business and medicine to help students learn from the successes and failures of practitioners in the field. This paper discusses the potential benefits of case studies in computer science and how case studies differ from projects. We describe our preliminary experience with developing a network simulation case study for an object-oriented CS2 course and present some of ideas for teaching such a case study through close coupling of lectures with laboratories. The teaching materials for this case study are available on the web.

1 Introduction

Case studies take many different forms, so we begin by discussing the essential elements of a case study. A case study should be based, to some extent, on a real-world problem. It should contain a significant amount of information about its subject and about the approach used to analyze and solve the problem. Ideally a case study should have data to be analyzed and interpreted.

How does a case study differ from a project? While both can be complex and reality-based, a significant part of a case study is the understanding and analysis of an existing solution. In contrast, a project generally emphasizes an initial design.

Case studies have been used to a limited extent in computer science programs, most notably in the teaching of software engineering [1]. A survey of recent SIGCSE conferences showed very few papers that could be classified as true case studies. A notable exception is the Advanced Placement Marine Biology Case Study [2].

Our primary motivation in developing a case study for CS2 was to give students experience with complex interactions of objects. We also wanted to show examples of good design in a realistic situation. This paper discusses a network simulation case study that we are developing and have used for the past 5 semesters in a Java-based CS2.

We selected a network simulation because we wanted a subject focus in computer science that would help students in later courses. Our operating systems and network courses use simulation at several levels to enhance student understanding [5], so an event-driven simulation seemed like a good candidate for development. We also thought that understanding event-driven simulation would reinforce student understanding of event-driven execution in their programs. The case study, described in more detail in the next sections, illustrates the use of abstract classes and methods. It has more complex class interactions than students see in their projects and uses queues and priority queues in a real application. The case study also allows students to analyze and present empirical data. The design makes the model easy to change and generalize.

We cover the case study about midway through the semester in our Java-based CS2 after covering stacks, queues, priority queues and interfaces. We introduce several of the supporting classes as examples during lecture in the weeks before starting the case study. The case study takes approximately two weeks of lecture and two weeks of laboratory.

A key element in introducing the case study was to integrate material across lecture, laboratory and homework. We discovered that this was much more difficult than we thought it would be. In the next section we briefly introduce the model. Section 3 introduces the basic program elements --- events, sources and servers. Section 4 discusses the simulation algorithm, while Section 5 explains the role of abstract classes in making a generalizable program. Section 6 discusses how the lecture and lab were integrated and some pitfalls that we encountered.

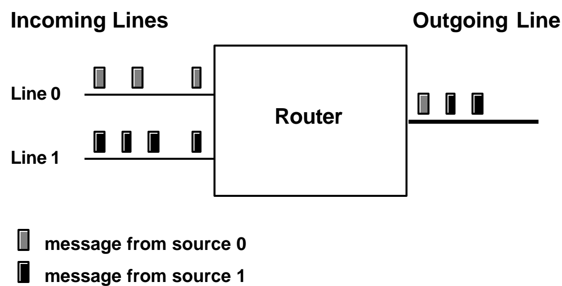

Figure 1 Model of a simple network router with 2 sources and one destination.

2 Understanding the model

The network case study focuses on the simulation of a network router that reads messages from several sources (incoming lines) and resends them on outgoing lines as shown in Fig. 1. The first step is a basic understanding of the model. Students encounter many new concepts: What is a message? What does it mean to route a message? How would you measure the traffic? What happens if too many messages come in?

When the model is introduced we talk about several analogous problems. For example, in an airport analogy the sources might be airport terminals and the messages airplanes. We ask the students what plays the role of the router at an airport (the control tower) and the outgoing line (the runway). As a graded exercise we ask students to come up with an additional real-world situation that could be described by the model of Fig. 1 We also discuss how the model might be generalized.

The first conceptual problem that students encounter is understanding how the basic picture of a router shown in Fig. 1 translates into events and arrival times. The router sees messages from Line 0 at times 10, 45 and 64. Line 1 produces messages at 10, 33, 50 and 69:

Line	Arrival Times	Interarrival Times
0	10, 45, 64	10, 35, 19
1	10, 33, 50, 69	10, 23, 17, 19

Fig. 2 shows a timeline for these incoming messages from the viewpoint of the router. The router is interested in how frequently the messages arrive, because it must be able to process them fast enough. The *interarrival* time is the time between successive messages. Line 0 produces messages with an average interarrival time of (10+35+19)/3 = 64/3 = 21.33, while Line 1 has a message interarrival time of (10+23+17+19)/4 = 69/4 = 17.25.

The message streams from Line 0 and Line 1 are combined in the router. The router sees a combined stream:

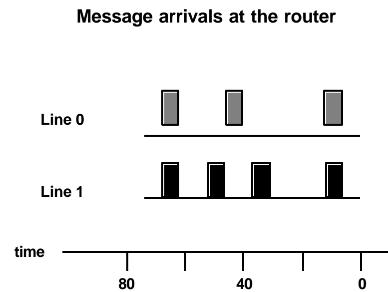

Figure 2 Message arrival times at the router.

Message arrival times: 10, 10, 33, 45, 50, 64, 69
Message interarrival times: 10, 0, 23, 12, 5, 14, 5
Average time between messages:
(10+0+23+12+5+14+5)/7 = 69/7 = 9.9

The router reads a message from an input line and copies it to an output line. To understand how a router works we have students trace some simple problems:

> Assume that when the router copies a message to its outgoing line, the outgoing line will be busy for 15 time units and can not accept another message. If the outgoing line is busy when it tries to copy the message, the router throws the message away. Draw a time line for the outgoing line. Which messages are lost?

The complexity of even a simple problem with few events provides a motivation for using simulation.

3 Representing events, sources and servers

Our initial versions of this case study introduced the simulation program at this point. However, we found that students were simply unable to understand how "events happened". Each time that we have taught the case study we have increased the number of preliminary exercises that practice with events. We have also increased the number of exercises and in class time spent on event sources (the incoming lines) and servers (the outgoing lines).

3.1 Events

In the simulation case study, events are represented by an `Event` class that implements the Java `Comparable` interface. We use `Event` as an example of an interface in CS1 and sort arrays of `Event` objects. In CS2 we use `Event` as a test case for insertion of objects on queues and priority queues prior to starting the case study. Appendix A.1 has an implementation of `Event`. We ask students to create objects corresponding to different events and to insert and remove event objects from a priority queue. We then discuss how events might be generated in a program. Appendix A.2 gives sample exercises.

3.2 Sources

The `IncomingLine` class shown in Appendix A.3 is the basic source of messages for the network. It uses a Poisson distribution provided in the `util` package of the course textbook [6], but it could be readily adapted to other distributions. We assume that students already understand pseudorandom number generation to some extent, because we use `Coin` and `Dice` classes extensively in CS1. `IncomingLine` objects have a `getNextArrival` method that returns an `Event` corresponding to the next message generated on its line.

As a lab exercise in week 1, students create an `IncomingLine` object and use it to generate a sequence of events. They calculate interarrival times and plot histograms of them. Students are also asked to generate an array of `IncomingLine` objects, place 10 events generated from each object in a priority queue, calculate interarrival times and generate a histogram. We ask students to modify this class to keep track of the average message interarrival time of the messages produced by the line. Appendix A. 4 has some examples of study questions for students to answer as part of the lab.

3.2 Servers

The `OutgoingLine` class is a simple server whose service is the amount of time it takes for the line to send a message. The router cannot put a message on an outgoing line when it is busy. In early parts of the case study we explore the idea that even if the average message interarrival time is long compared with the service time, some messages will still be lost. In later parts of the assignment we explore the idea of using a buffer (queue) to smooth out arrivals and eliminate lost messages.

4 What is simulation and why do it?

The major stumbling block of the case study is for students to understand how event-driven simulation works. The word *simulate* means to imitate or have the same effect as. Computer simulation imitates an activity in the real world using a computer program in order to understand how the real-world process behaves. We emphasize that the advantage of computer simulation is that you can easily change the system to see the impact of modifications on the result. We also use analogies with more familiar examples such as traffic flow on highways and airports. Assuming that we keep track of the events using a priority queue, the basic simulation algorithm is:

- generate the initial events and insert in priority queue
- while the priority queue is not empty and the simulation has not been stopped:
  - get and delete next event from the priority queue
  - update current time to the event's time
  - if the event is `STOP`, break
  - process the event (possibly generating new events and inserting them on the priority queue)

At this point we work through several small examples by hand. We also look at a direct implementation of a simulation with one incoming line.

5 Using abstract classes in the design

During our first few attempts, we presented the case study, by showing students the final design and working through it. We felt that students didn't really understand the simulation concepts using this approach, so we started a more bottom-up approach -- developing an understanding of the basic components (events, sources and servers) before introducing abstract classes.

We motivate the use of an abstract class by the observation that the basic structure of all simulation models is similar--they only differ in the details of how they process events. We introduce the `SimulationModel` abstract class to represent the common elements:

```
public abstract SimulationModel {
  public SimulationModel(String title) { … }
  public final void addEvent( Event e) { … }
  public final int getEvents( ) { … }
  public final double getTime( ) { … }
  public final String getTitle( ) { … }
  public abstract void initializeEvents( );
  public abstract void
                    processEvent(Event e);
  public final void run(double stoppingTime)
                       { … }
  public final void setSimulationStopped( )
                       { … }
}
```

The `final` methods are complete and cannot be changed by any class that extends (is-a) `SimulationModel`. The `initializeEvents` and `processEvent` are abstract, so `SimulationModel` is abstract. `SimulationModel` objects cannot be instantiated. Appendix A.5 gives an implementation of the run method of `SimulationModel`. We carefully trace this method with different events.

We also explain that a subclass of `SimulationModel` is needed for each problem that we want to solve. The subclass only needs `initializeEvents` and `processEvents` methods, because the superclass has all of the simulation machinery. Appendix A.6 shows an example of a subclass `SingleIn` that consists of a single incoming line. Students are asked to add the code to update some of the statistical information. We also work through examples in lecture and/or lab that have a single incoming line and a single outgoing line, multiple incoming and outgoing lines and the addition of buffering at the router. Students are asked to a run series of experiments and plot the results. They are also asked to make hypotheses about the expected results and relate their hypotheses to the actual results. The materials for the case study are available on the web [3].

6 Experiences and additional thoughts

Many of our students expressed the feeling that reading and understanding existing code was difficult and time-consuming. One of the authors (KD) was a student presenter [4] for CS2 during the five semesters reported on in this paper. His comment below echoes the sentiments expressed by many other students:

> "As a student, my initial confusion was only overcome after a considerable amount of work with the models increased my familiarity with the source code. Many of my contemporaries had difficulty understanding the functions of the source code and how to alter the model to execute new requirements described in the lab recitations. These problems were addressed in three ways. First, the recitations were changed to include requirements for the students to test each of the classes individually, before they were integrated into an overall model. This helped the students understand the source code in each individual class. Second, a number of study questions were added to the recitations to focus on specific issues that needed to be emphasized. Third, and finally, minor alterations to the source code were made to make it clearer and more understandable to the students. The combination of these three additions dramatically improved the students understanding of the source code and their ability to work with the simulation models. As a presenter, I found the improved version of the lab recitation to be a much better process. The students understood the simulation better and were more comfortable working with the classes to incorporate new changes."

We found that assigning simple client based-exercises encouraged the students to focus on the behavior of each object before attempting the interaction of the objects. Students said that the study questions for each section helped them to understand important concepts that they might otherwise have missed.

Variations on the network router provide a wealth of lecture material for the instructors, and the lab exercises provide hands-on experience for the student. However, some students felt that there was a duplication of material being taught in the lecture and lab. To improve the coordination, we had regular meetings with the lab presenters to coordinate the lessons taught in the lab with the lecture material.

This case study is not for the faint of heart, but we believe that the extra time and effort needed to teach it are justified. Our students demonstrate knowledge and skills over a variety of topics that otherwise they would not have been exposed to. We also observed that subsequent assignments of this size and difficulty are not so overwhelming.

Acknowledgments

This work was made possible by a NASA Grant NAG2-6040. We would also like to thank the presenters, tutors and TAs for their help in making the course work.

References

[1] Clancy, M. J. and Linn, M. C., Case studies in the classroom. SIGSCE'92 (Feb 1992) ACM Press, 220-224.

[2] Brady, A., Clancy, M. J., and Larson, K., Seminar: Introduction to the Marine Biology Case Study. SIGSCE '2001 (Feb, 2001), ACM Press, 427.

[3] See http://vip.cs.utsa.edu/NASACurriculum/papers/ for the case study and the associated teaching materials or http://vip.cs.utsa.edu/NASACurriculum for the main project homepage.

[4] Robbins, K., Key, C., Dickinson, K. and Montgomery, J., Solving the CS1/CS2 lab dilemma: students as presenters in CS1/CS2 laboratories. SIGCSE'2001 (Feb, 2001), ACM Press, 164-168.

[5] Robbins, S. and Robbins, K., Empirical exploration in undergraduate operating systems. SIGCSE '99 (Mar, 1999) ACM Press, 311-315.

[6] Weiss, M. Data Structures and Problem Solving Using Java, 2nd Ed, Addison Wesley, 2001.

Appendix A

Appendix A.1: The Event Class

```
public class Event implements Comparable {
  public static final int STOP = 0;
  public static final int MSG_ARRIVAL = 1;
  public static final int MSG_SENT = 2;
  private int who; // Who did it
  private int what; // What they did
  private double when; // When they did it

  public Event(int ID, int type,
               double time) {
     who = ID;
     what = type;
     when = time;
  }

  public int getWho( ) {return who;}
  public int getWhat( ) {return what;}
  public double getWhen( ) {return when;}

  public int compareTo (Object rhs)
              throws ClassCastException{
    double rhsTime = ((Event)rhs).getWhen();
    return when < rhsTime ? -1:
           rhsTime < when ? 1: 0;
  }
}
```

Appendix A.2: Sample Exercises for Event

- Add a `toString` method to the `Event` class.
- Generate client code to create three events with:

who	what	when
0	1	10
1	1	13
2	1	11

  Print each event using the `toString` method.
- Generate client code to create a `PriorityQueue` named `p`. Insert the three events in `p` using `insert`.
- Write a loop that deletes (use `deleteMin`) and prints each event until the `PriorityQueue p` is empty. Which data field in the `Event` class determines the order in which events are removed from `p`?
- Create an `Event` with a type that is not identified (e.g., `what` is 3). What happens? How would you add a new type of event to the `Event` class?

Appendix A.3: The IncomingLine Class

```
public class IncomingLine {
  private double interarrival;
  private int ID;
  private int totalMessages;
  private Random randProc;
  private double time;

  public IncomingLine(int theID,
    double start, double arrivalParm) {
    interarrival = arrivalParm;
    ID = theID;
    totalMessages = 0;
    randProc = new Random(ID + 1);
    time = start;
  }

  public int getID( ) {return ID;}
  public double getTime( ) {return time;}
  public int getTotalMessages ( )
                   {return totalMessages;}

  public Event getNextArrival( ) {
    time += randProc.poisson(interarrival);
    totalMessages++;
    return new
        Event(ID, Event.MSG_ARRIVAL, time);
  }
}
```

Appendix A.4: Sample Exercises for IncomingLine

- Write client code to create an `IncomingLine` object with an `arrivalParm` of 30.0, a start time of 0, and an `ID` of 0. Use the `getNextEvent` method of the object to generate 100 messages and calculate the average interarrival time (i.e., the average time between messages). How do you expect the average interarrival time to be related to the `arrivalParm`? Try several different values of the `arrivalParm` to confirm your hypothesis.
- How can `IncomingLine` keep track of the actual average interarrival time of the messages it generates? Add a `getAverageInterarrival` method that returns the average interarrival time of the messages that the line has generated so far.
- Write client code to create 3 `IncomingLine` objects with an `arrivalParm` of 30.0 and IDs of 0, 1, and 2, respectively. Generate 5 messages on the first line, 4 messages on the second line and 3 messages on the third line. Print each message interarrival time and the average interarrival time for each line.
- Using the interarrival times generated in your project make a list of arrival times for each line (5 for line 0, 4 for line 1, 3 for line 2). Using the arrival times for all lines, calculate the average time between messages.

Appendix A.5: The SimulationModel Class run method

```
final public void run(double stopTime) {
  Event e = null;
  initializeSimulation(stopTime);
  initializeEvents();
  while(!simulationStopped &&
        !eventSet.isEmpty()){
     e = (Event) (eventSet.deleteMin( ));
     numberEvents++;
     currentTime = e.getWhen( );
     processEvent(e);
  }
}
```

Appendix A.6: The SingleInModel Class

```
public class SingleInModel extends
                       SimulationModel {
  private IncomingLine inLine;
  private int totalMessages;
  private double lastArrivalTime;
  private double totalArrivalTime;

  public SingleInModel (double arrival){
    super("SingleIncomingModel");
    inLine = new IncomingLine(1,0,arrival);
  }

  public void initializeEvents( ) {
    addEvent(inLine.getNextArrival());
  }

  public void processEvent(Event e) {
    switch (e.getWhat()) {
      case Event.MSG_ARRIVAL:
        addEvent(inLine.getNextArrival());
        break;
      case Event.STOP:
        setSimulationStopped();
        break;
      default:
        System.err.println(e +
            " {unrecognized event}");
       break;
    }
  }
}
```