Discrete-Event Simulation of Queueing Systems in `Java`: The `jsimulation` and `jqueues` Libraries

Guided Tour

Jan de Jongh

Release 5.2, 5.3-SNAPSHOT

DRAFT

Print/Release Date 20180503

Preface

This document describes the publicly available `jsimulation` and `jqueues Java` libraries for discrete-event simulation of queueing systems. Personally, I (merely) scratched the surface of queueing theory at Twente University back in the eighties, while working on my Master's Thesis on an operating system for *transputers*. Transputers are fast RISC processors with multiple on-chip communication links; back then, they were envisioned to become the building blocks of future massively parallel computer systems. Since our main applications of interest were in robotics, I attempted basic queueing theory in an attempt to find hard real-time response-time guarantees, in order to meet physical-world, mostly safety-related, deadlines.

During the largest part of the nineties, I worked on my PhD at Delft University of Technology. This time, I got to study queueing systems modeling *distributed computing systems*, which by then had overtaken parallel systems in terms of scientific interest. The main purpose of the research was to devise and analyze scheduling strategies for dividing in space and in time the computing resources of a (closed) distributed system among groups of users, according to predefined policies (named *share scheduling* at that time). In order to gain quantitative insight, I used the classic `DEMOS` (Discrete Event Modeling On Simula) software running on the `SIMULA` programming language. I made several modifications and extensions to the software, in order for it to suit my needs. For instance, it lacked support for so-called *processor-sharing* queueing disciplines in which a server ("processor") distributes at any time its service capacity among (a subset of) jobs present. In addition, I needed a non-standard set of statistics gathered from the simulation runs. In the end, both `DEMOS` and `SIMULA` itself proved flexible enough to study the research questions.

Like `DEMOS`, the `jsimulation` and `jqueues Java` software packages described in this book feature discrete-event simulation of queueing systems. The libraries are, as a combo, somewhat comparable to the `DEMOS`, yet there are important differences nonetheless. For instance, the libraries focus exclusively at *algorithmic* modeling of queueing systems and job visits; they do not cover additionally required features like sophisticated random-number generation, probability distributions, gathering and analyzing statistics, and sophisticated reporting; features all integrated in `DEMOS`. In that sense, `DEMOS` is a more complete package. On the other hand, the packages feature a larger range of queueing-system types, and, for instance, a model for constructing new queueing systems through *composition* of other queues. In addition, much care has been put into the *atomicity* of certain events, which allows for a wider range of *queue invariants* supported. Despite these differences, `DEMOS` has been a major inspiration in the design and implementation of `jsimulation` and `jqueues`.

For my current employer, TNO, I have performed, over the past decade (or even decades?), many simulation studies in `Java` related to the vehicle-to-vehicle communications in (future) Intelligent Transportation Systems, studying, for instance, position dissemination over CSMA/CA wireless networks for Cooperative Adaptive Cruise Control and Platooning. At some point, I realized that it would be feasible to extract some useful and stable `Java` libraries from my ever increasing software repositories, and release them into the public domain. So, in a way, the libraries can be considered "collateral damage" from a variety of projects.

Chapter 1

Introduction

Queueing systems deal with the general notion of *waiting* for (the completion of) "something". They are ubiquitously and often annoyingly present in our everyday lives. If there is anything we do most in life, it is probably *waiting* for something to happen (finally winning a non-trivial prize in the State Lottery after paying monthly tickets over the past thirty years), arrive (the breath-taking dress we ordered from that webshop against warnings in the seller's reputation blog), change (the reception of many severely bad hands in the poker game we just happened to ran into), stop (the constant flipping into red of traffic lights while we are just within breaking distance in our urban environment), or resume (the heater that regularly happens to have a mind of its own during winter months).

Queueing systems also appear in computer systems and networks, in which they schedule available shared resources like processors, memory, and network ports among clients like computing applications. Or in wireless communications, where so-called 'listen-before-talk' access protocols (CSMA/CA) as used in wireless Local Area Networks monitor the received power level at the input stage in order to assess whether the transmission medium is idle before attempting to transmit. Or in automated production lines, where a partial product is routed to visit several service stations in sequence, each of them performing a specific task to the product.

Perhaps surprisingly given their wide variety in terms of applications, queueing systems usually share a common concept: a set of objects we will call *jobs* has to visit a set of objects we will call *queues*, in order to get something done. Depending on the complexity of the task to be performed, on the service capacity of the queue, and on the available competition among jobs, such a visit may vary in length (i.e., in sojourn time). This perhaps explains the great interest from the mathematical community in *queueing theory*: One often needs only a handful of variables and assumptions in order to model a wide range of applications. In most cases, the effects of these assumptions are modeled with suitable stochastic processes.

Despite great results in deriving closed-form analytic expressions for many queueing models, many more others are mathematically intractable. In order to gain quantitative insight into these models, one often resorts or needs to resort to *discrete-event simulation*, appropriately modeling queue scheduling behavior, and subjecting it to a workload consisting of jobs with appropriate parameters as to the amount of work each job requires, and the time between consecutive job arrivals. Even though discrete-event simulation does not provide closed-form solutions, they are often very handy and capable of, for instance, quantitative comparisons between various scheduling strategies.

This document introduces `jsimulation` and `jqueues`, open-source `java` software libraries for discrete-event simulation of queueing systems. Its main purpose is to expose you to the most important concepts in the libraries, and to get you going with your simulation studies. By no means is this document complete in its description of `jsimulation` and `jqueues`, nor is it intended to be, and for more detailed information we refer the reader to the "JQueues

Reference Manual"[1] if you need precise specification of the libraries, and to the "JQueues Developer Manual"[2] if you want or need to extend either library (e.g., to add your own queueing discipline).

In Section 2 of the present document we provide installation (and build) instructions, and in Section 3 we present our "Hello World" example. In subsequent sections, in rather random order, we provide additional details and examples on the use of both libraries; attempting to allow linear reading. However, this is a living document and sections are added on demand and when time permits.

Any feedback on the clarity and/or correctness of the text is highly appreciated. Please use the *Issues* section on `github` to that purpose[3].

[1] The JQueues Reference Manual is currently being written, and will be available as an e-Book.
[2] The JQueues Developer Manual is currently being written, and will be available as an e-Book.
[3] See `https://github.com/jandejongh/jqueues-guided-tour`.

Chapter 2

Installation

2.1 The jsimulation and jqueues Libraries

In order to use jsimulation and jqueues, you have to install them first, which requires an Internet connection. The first public releases of jqueues and jsimulation have version number 5.2.0; they have been released under the Apache v2.0 license. From that version number onward, both libraries are distributed as Maven projects available from github.com and the Maven Central Repository (whichever suits you).

Since both jsimulation and jqueues are libraries and hardly support stand-alone operation, we assume that you intend to install them both as dependencies to your own project. You have several options, but the two most obvious ones are:

- Install the libraries from github, open them as Maven *projects* in your IDE and add them as dependencies to your own project. If you use Maven yourself for the latter, you only have to add the dependency on jqueues in the pom.xml. (You do not have to add jsimulation because Maven does this automatically for you.)

- Create your own Maven project and add jqueues as a dependency, taken from the Maven Central Repository.

In both cases, you will need maven installed and properly configured on your system. It is also highly recommended to install maven support in your IDE, so that it can directly open maven projects.

In the first case, you need git as well, and you should clone both libraries from github as shown below:

- $ git clone https://www.github.com/jandejongh/jsimulation

- $ git clone https://www.github.com/jandejongh/jqueues

Note that jsimulation and jqueues can only be built against Java 1.8 and higher.

In the second case, add the XML fragment shown in Listing 2.1 to the dependencies section in your pom.xml. Please make sure that you double-check the version number in the XML file[1]. The second case is safer as it uses stable, frozen, versions of the libraries released to Maven Central. These releases are signed and cannot be changed without increasing the version number.

[1] You may want to verify the latest stable release number from either github or Maven Central. This Guided Tour applies to release 5.2.0 and beyond.

Listing 2.1: The `dependency` section for `jqueues` in a `pom.xml`.

```
<dependency>
  <groupId>org.javades</groupId>
  <artifactId>jqueues</artifactId>
  <version>5.2.0</version>
  <scope>compile</scope>
  <type>jar</type>
</dependency>
```

2.2 Version Numbering

For both libraries, we use three-level version numbering:

- The third, lowest, level is reserved for bug fixes, `javadoc` improvements and code (layout) "beautifications".
- The second, middle, level is reserved for functional extensions that do not break existing code (with the same major version number). Think of adding another queue, job or listener type.
- The third, major, level is reserved for changes to the core interfaces and classes that are likely to break existing code.

This implies that you can (should be able to) always "upgrade" to a later version from Maven Central as long as the major number remains the same. Upgrading from `github.com` requires a bit of care, as the latest version may not be stable yet.

Despite the fact that we take utmost efforts to *not* break existing code with upgrades of middle and minor version numbers, we cannot always avoid this. For instance, we may realize that a method should be `final` or `private` and attempt to fix that in an apparent innocent update, but you may have overridden (or used) that particular method already in your code to suit your own purposes. Needless to say, we did not expect you to override (or just use) that particular method in your code, just as well as you did not expect that you were not supposed to do so. But in the end, your code may not be compile-able after the upgrade. In order to avoid this, we recommend that you

- Prefer interface methods rather than specific ones from classes, since the chance that we consider updates of the interface as being "minor" is virtually nil.
- Only override methods for which the `javadoc` explicitly states that they are intended to be overridden.

2.3 The `jqueues-guided-tour` Project

All example code shown in this document is available from the `jqueues-guided-tour` project on `github`. The code is organized as a Maven project. In addition to the example code, it also contains all the source files (LaTeX and other) to the present document. Bear in mind, though, that the documentation and example code in `jqueues-guided-tour` are both released under a more restrictive license than `jsimulation` and `jqueues`. In short, you are allowed to use the documentation and example code to whatever purpose. You may also redistribute both in unmodified form. However, redistributing *modified* versions of either or both of them requires the explicit permission from the legal copyright holder.

Chapter 3

Hello World: FCFS

In this section, we introduce our "Hello World" application for `jqueues`[1], consisting of a `FCFS` queue subject to arrivals of jobs with varying required service times.

In order to perform a simulation study in `jqueues`, the following actions need to be taken:

- The creation of an event list;
- The construction of one or more queues attached to the event list;
- The selection of the method for listening to the queue(s);
- The creation of a workload consisting of jobs and appropriately scheduling it onto the event list;
- The execution of the event list;
- The interpretation of the results, typically from the listener output.

Without much further ado, we show our "Hello World" example in Figure 3.1. We first create a single event list of type `DefaultSimEventList` and a `FCFS` queue attached to the event list (by virtue of the argument of `FCFS`'s constructor). On the queue, we register a newly created `StdOutSimEntityListener`, issuing notifications to the standard output. Note that queues and jobs are so-called *entities*; these are the relevant objects with state subject to event invocation. Subsequently, we create ten jobs named "0", "1", "2", ..., scheduled for arrival at the queue at $t = 0$, $t = 1$, $t = 2$, ..., respectively, and set their respective service times. We then schedule each job arrival on the event list. Finally, we "run" the event list, i.e., let it process the arrivals.

Listing 3.1: A simple simulation with a single FCFS queue and ten jobs.

```
final SimEventList el = new DefaultSimEventList (0);
final SimQueue queue = new FCFS (el);
queue.registerSimEntityListener (new StdOutSimEntityListener ());
for (int j = 0; j < 10; j++)
{
  final double jobServiceTime = (double) 2.2 * j;
  final double jobArrivalTime = (double) j;
  final String jobName = Integer.toString (j);
  final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
  SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
el.run ();
```

The event list type `DefaultSimEventList` will suffice for almost all practical cases, but it is essential to note already that a *single* event-list instance is typically used throughout

[1] In this Chapter, whenever we refer to `jqueues`, we silently assume that `jsimulation` is installed as well.

any simulation program. Its purpose of the event list is to hold scheduled *events* in non-decreasing order of *schedule time*, and, upon request (in this case through `el.run`), starts processing the scheduled events in sequence, invoking their associated *actions*. In this case, the use of events remains hidden, because jobs are scheduled through the use of utility method `scheduleJobArrival`. The zero argument to the constructor denotes the simulation start time. If you leave it out, the start time defaults to $-\infty$.

Our queue of choice is First-Come First-Served (FCFS). The constructor takes the event list `el` as argument. The queueing system consists of a queue with infinite places to hold jobs, and a single server that "serves" the jobs in the queue in order of their arrival. Once a queue has finished serving the (single) job, the job *departs* from the system.

So how long does it take to serve a job? Well, in `jqueues`, the default behavior is that a queue requests the job for its *required service time*. In the particular case of `DefaultSimJob` (there are many more job types), we provide a fixed service time (at *any* queue) upon creation through the third argument of the constructor.

The first argument of the `DefaultSimJob` is the event list to which it is to be attached. For jobs (well, at least the ones derived from `DefaultSimJob`), it is often safe to set this to `null`, although we could have equally well set it to `el`. However, *queues must always be attached to the event list*; a `null` value upon construction will throw an exception.

The (approximate) output of the code fragment of Listing 3.1 is shown in Listing 3.2 below. Remarkably, the listing only shows two types of notifications, viz., `UPDATE` and `STATE_CHANGED` ↪ , the latter of which can hold multiple "sub"-notifications. Each notification outputs the name of the listener, the time on the event list, the queue (entity) that issues the notification, the notification's actual "major" type (`UPDATE` or `STATE_CHANGED`) and, if present, the sub-notifications.

Apart from the `STATE CHANGED`, `UPDATE` and `START_ARMED` lines in the output, the notifications pretty much speak for themselves. We even get notified when jobs start service (`START`). The `START_ARMED` notifications refer to state changes in a special `boolean` attribute of a queue named its `StartArmed` property. Since you will hardly need it in practical applications, we will not delve into it, but it is crucial for the implementation of certain more complex (composite) queueing systems. Suffice it to say that the `StartArmed` property *in this particular case* signals whether the queue is idle.

The two top-level notification types, `UPDATE` and `STATE CHANGED` are essential. Upon every change to a queue's state, the queue is obliged to issue the fundamental `STATE CHANGED` notification, exposing the queue's new state (including its notion of time). The `UPDATE` notification has the same function, but it is fired *before* any changes have been applied, thus revealing the queue's *old* state, including the time at which the old state was obtained. Hence, every `STATE CHANGED` notification *must* be preceded with an `UPDATE` notification. The `UPDATE` notification is crucial for the implementation of statistics (among others).

The use of `STATE_CHANGED` notifications may appear strange at first sight as many other implementations would report each of the sub-notifications individually. However, an important aspect of a queue's contract is that *it must report state changes atomically in order to meet queue invariants*. This means that listeners, when notified, will always see the queue in a consistent state, i.e., in a state that respects the invariant(s). This is one of the (we think) most distinguishing features of `jqueues`. Going back to our example: An important invariant of FCFS and many other queueing systems is that there cannot be jobs waiting in queue while the server is idle. It is easy to see that individual notifications for `ARRIVAL` and `START` would lead to violations of this invariant: Suppose that a job arrives at an idle FCFS queue. Using individual notifications, the queue has no other option than to issue a `ARRIVAL` notification immediately followed by a `START`. In between both, the queue would expose a state that is inconsistent with the invariant because the server is idle (the job has not started yet), while

Listing 3.2: Example output of Listing 3.1.

```
StdOutSimEntityListener t=0.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[0]@FCFS]
  => START [Start[0]@FCFS]
  => DEPARTURE [Dep[0]@FCFS]
StdOutSimEntityListener t=1.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=1.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[1]@FCFS]
  => START [Start[1]@FCFS]
  => STA_FALSE [StartArmed[false]@FCFS]
StdOutSimEntityListener t=2.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=2.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[2]@FCFS]
StdOutSimEntityListener t=3.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=3.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[3]@FCFS]
StdOutSimEntityListener t=3.2, entity=FCFS: UPDATE.
StdOutSimEntityListener t=3.2, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[1]@FCFS]
  => START [Start[2]@FCFS]
StdOutSimEntityListener t=4.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=4.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[4]@FCFS]
StdOutSimEntityListener t=5.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=5.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[5]@FCFS]
StdOutSimEntityListener t=6.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=6.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[6]@FCFS]
StdOutSimEntityListener t=7.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=7.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[7]@FCFS]
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS: UPDATE.
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[2]@FCFS]
  => START [Start[3]@FCFS]
StdOutSimEntityListener t=8.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=8.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[8]@FCFS]
StdOutSimEntityListener t=9.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=9.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[9]@FCFS]
StdOutSimEntityListener t=14.200000000000001, entity=FCFS: UPDATE.
StdOutSimEntityListener t=14.200000000000001, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[3]@FCFS]
  => START [Start[4]@FCFS]
StdOutSimEntityListener t=23.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=23.0, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[4]@FCFS]
  => START [Start[5]@FCFS]
StdOutSimEntityListener t=34.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=34.0, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[5]@FCFS]
  => START [Start[6]@FCFS]
StdOutSimEntityListener t=47.2, entity=FCFS: UPDATE.
StdOutSimEntityListener t=47.2, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[6]@FCFS]
  => START [Start[7]@FCFS]
StdOutSimEntityListener t=62.60000000000001, entity=FCFS: UPDATE.
StdOutSimEntityListener t=62.60000000000001, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[7]@FCFS]
  => START [Start[8]@FCFS]
StdOutSimEntityListener t=80.20000000000002, entity=FCFS: UPDATE.
StdOutSimEntityListener t=80.20000000000002, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[8]@FCFS]
  => START [Start[9]@FCFS]
StdOutSimEntityListener t=100.00000000000001, entity=FCFS: UPDATE.
StdOutSimEntityListener t=100.00000000000001, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[9]@FCFS]
  => STA_TRUE [StartArmed[true]@FCFS]
```

there is a job in its waiting queue. Note that another invariant of FCFS is that it cannot be serving jobs with zero required service time. This explains the arrival, start and departure sub-notifications for job 0 are all in a single atomic `STATE_CHANGED`.

This concludes our "Hello World" example. There is obviously a lot more to tell, but the good news is that our example has already revealed the most important concepts of `jqueues` like the event list, events, entities, queues, jobs, listeners and notifications. The remaining complexity is in the richness and variation of these basic concepts.

Chapter 4

Events, Actions and the Event List

This chapter describes the event and event-list features that are available from the `jsimulation` package. Note that `jsimulation` is a dependency of `jqueues`. In most usage scenarios, there is no need to directly manipulate events or the event-list; the preferred method is to use *utility* methods for that. However, in order to describe in more detail the models of entities, jobs and queues, a basic understanding of what goes on under the hood of a `DefaultSimEventList` is very helpful.

4.1 Creating the Event List and Events

At the very heart of every simulation experiment in `jqueues` is the so-called *event list*. The event list obviously holds the events, keeps them ordered, and maintains a notion of "where we are" in a simulation run. Together, an event list and the events it contains define the precise sequence of actions taken in a simulation. The code snippet in Listing 4.1 shows how to create an event list and schedule two (empty) events, one at $t_1 = 5.0$ and one at $t_2 = 10$, and print the resulting event list on `System.out`. In `jsimulation`, the event list is of type `SimEventList`; events are of type `SimEvent`, respectively. Since both of them are Java *interfaces*, you need implementing classes to instantiate them: `DefaultSimEventList` for an event list; `DefaulSimEvent` for an event; typically you need a single event list and numerous events.

Listing 4.1: Creating the event list and populating it with events.

```
final SimEventList el = new DefaultSimEventList (-5);
final SimEvent e1 = new DefaultSimEvent (5.0);
final SimEvent e2 = new DefaultSimEvent (10.0);
el.add (e1);
el.add (e2);
el.print ();
el.run ();
System.out.println ("Finished!");
```

As explained in the previous chapter, the `double` argument in the `DefaultSimEventList` constructor is the initial time on the event list, its so-called *default reset time*. The `double` argument in the `DefaultSimEvent` constructor (of which there are several) is the *schedule time* of the event on the event list. Events, once created, are scheduled on the event list through the `add` method; the event list stores the events until use and maintains the proper order between them. The output of the code snippet is shown in Listing 4.2[1]:

By virtue of the call to `el.print`, the output shows the name of the event list (as obtained from its `toString` method) and the current time (-5) in the first row, and then the events in

[1] We may have improved the layout in the meantime.

Listing 4.2: Output of Listing 4.1.

```
SimEventList EventList[t=-5.0], class=DefaultSimEventList, time=-5.0:
t=5.0, name=No Name, object=null, action=null.
t=10.0, name=No Name, object=null, action=null.
Finished!
```

the list in the proper order. Beware that the event-list is printed before the `el.run` statement; it would be empty afterwards.

Perhaps surprisingly, in `jsimulation`, the schedule time is actually held on the event, *not* on the event list. Also, a `SimEventList` is inheriting from `SortedSet` from the Java Collections Framework. These choices have the following consequences:

- Each `SimEvent` can be present *at most once* in a `SimEventList`. You cannot reuse a single event instance (like a job creation and arrival event) by scheduling it multiple times on the event list. Instead, you must either use separate event instances, or reschedule the event the moment it leaves the event list.
- You cannot (more precisely, *should not*) modify the time on the event while it is scheduled on an event list.
- You always have access to the (intended) schedule time of the event, without having to refer to an event list (if the event is scheduled at all) or use a separate variable to keep and maintain that time.
- The events must be equipped with a *total ordering* (imposed by `SortedSet`) and distinct events should not be equal (imposed by us). This means that for each pair of (distinct) events scheduled on a `SimEventList`, one of them is always strictly larger than the other (in the ordering, they cannot be "equal").
- If two or more events with identical schedule times are scheduled on a single event list, their relative order needs to be determined by other means than their schedule time. The `DefaultSimEventList` uses a random-number generator to break such ties. If, for some reason, you want to maintain *insertion order*, please have a look at `DefaultSimEventList_IOEL` ↪ . Note that `IOEL` stands for Insertion Order Event List. But be warned: all (concrete) queue types in `jqueues` are specified against random ordering of simultaneous events.

Clearly, there is a lot more to say about simultaneous events, and about the reasons we chose for their random ordering while processing them, but we defer a detailed discussion for a later section. Nonetheless, it is important to realize that while an event say `e1` is being processed at some time t, any other event say `e2` scheduled at the same time on the event list is *always* processed after completion of `e1`. Even if `e1` itself actually schedules `e2`. In other words, `jsimulation` does *not* support the concept of *event preemption*, and the action of an event (see below) is always processed atomically. This implies that it will not work to use the event list (1) to get something done "immediately after" the completion of an event, (2) to do something "when all other events at t" are done", and (3) to process an event `e2` while processing an event `e1` and then returning to the original event `e1`.

4.2 Events

The output in Listing 4.2 shows four properties of a `SimEvent`:

- `Time`: The (intended) schedule time of the event (default $-\infty$).

- `Name`: The name of the event, which is only used for logging and output (default "No Name").
- `Object`: A general-purpose object available for storing information associated with the event (`jsimulation` nor `jqueues` uses this field; its default value is `null`).
- `EventAction`: The action to take, a `SimEventAction` (default `null`), described in the next section.

Each property has corresponding getter and setter methods on every `SimEvent`. In addition, `DefaultSimEvent` features multiple constructors that allow direct setting all or some of these properties upon construction.

4.3 Actions

A `SimEventAction` defined what needs to be done by the time an event is *executed* or *processed*. In Java terms, a `SimEventAction` is an interface with a single abstract method which is invoked when the event is processed, in other words, it is a `FunctionalInterface` that can be used in lambda expressions. We show its declaration in Listing 4.3.

Listing 4.3: The `SimEventAction` interface.

```
@FunctionalInterface
public interface SimEventAction<T>
{

  /** Invokes the action for supplied {@link SimEvent}.
   *
   * @param event The event.
   *
   * @throws IllegalArgumentException If <code>event</code> is <code>null</code>.
   *
   */
  public void action (SimEvent<T> event);

}
```

There are several ways to create a `SimEventAction` but nowadays, by far the easiest is to use lambda expressions, as shown in Listing 4.4. Note that we are now using the full `DefaultSimEvent` constructor, passing a name, and supplying a `SimEventAction` through a lambda expression. The generated output is shown in Listing 4.5. Note that we replaced the package and class identification of the action with `X` for formatting purposes.

Listing 4.4: Creating and using `SimEventActions`.

```
final SimEventList el = new DefaultSimEventList (0);
final SimEvent e = new DefaultSimEvent ("My First Real Event", 5.0, null, ((event) ->
  {
    System.out.println ("Event=" + event + ", time=" + event.getTime () + ".");
  }));
el.add (e);
el.print ();
el.run ();
el.print ();
```

4.4 Processing the Event List

Once the events of your liking are scheduled on the event list, you can start the simulation by *processing* or *running* the event lists. Processing the event list will cause the event list to equentially invoke the actions attached to the events in increasing-time order. There are several ways to process a `SimEventList`:

Listing 4.5: Example output of Listing 4.4.

```
SimEventList EventList[t=0.0], class=DefaultSimEventList, time=0.0:
  t=5.0, name=My First Real Event, object=null, action=X$$Lambda$1/1826771953@65ab7765.
Event=My First Real Event, time=5.0.
SimEventList EventList[t=5.0], class=DefaultSimEventList, time=5.0:
  EMPTY!
```

- You can process the event list until it is empty with the `run` method.
- You can process the event list until some specified (simulation) time with the `runUtil` method.
- You can *single-step* through the event list with the `runSingleStep` method.

You can check whether an event list is being processed through its `isRunning` method.

While processing, the event list maintains a *clock* holding the (simulation) time of the current event. You can get the time from the event list through `getTime` nethod, although you can obtain it more easily from the event itself. You can insert new events while it is being processed, *but these events must not be in the past.* Once the event list detects insertion of events in the past, it will throw and exception.

Note that processing the event list is thread-safe in the sense that all methods involved need to obtain a *lock* before being able to process the list. Trying to process an event list that is already being processed from another thread, or from the thread that currently processes the list, will lead to an exception. Note that currently there is no safe, atomic, way to process an event list on the condition that is is not being processed already. Though you can check with `isRunning` whether the list is being processed or not, the answer from this method has zero validity lifetime.

4.5 Utility Methods for Scheduling Events

A `SimEventList` supports various methods for directly scheduling events and actions without the need to generate both the `SimEvent` *and* the `SimEventAction`. In most cases, the availability of one of the objects suffices. In Table 4.1 we show the most common utility methods for scheduling on a `SimEventList`. The use of these utility methods is highly preferred over direct manipulation of the underlying `SortedSet` interface, because we (may) intend to delete the `SortedSet` dependency in future releases altogether.

Note that `E` refers to the so-called *generic-type argument* of `SimEventList`. The prototype is `SimEventList<E extends SimEvent>`. The use of the generic type `E` allows you to restrict the use of a `SimEventList` to certain types of `SimEvent`s, but for now `E` can be simply read as a `SimEvent`.

For any of the utilty methods that take a `SimEventAction` as argument, a new `SimEvent` is created on the fly, and returned from the method. Upon return from these methods, the newly created event has already been scheduled, and you *really* should not schedule it again.

So, how to *remove* events and actions from the event list? Well, since `SimEventList` implements the `Set` interface for `SimEvent` members, removing an event `e` from an event list `el` is as simple as `el.remove (e)`. However, the preferred method is `el.cancel (e)`.

4.6 Summary

The fundamental concepts in `jsimulation` are:

Table 4.1: Utility methods for scheduling on a `SimEventList`.

Utility methods for scheduling on `SimEventList`
void schedule (E) Schedules the event at its own time[2].
boolean cancel (E) Cancels (removes) a scheduled event, if present.
void schedule (**double**, E) Schedules the event at given time.
reschedule (**double**, E) Reschedules (if present, else schedules) the event at given new time.
E schedule (**double**, SimEventAction, String) Schedules the action at given time with given event name.
void scheduleNow (E) Schedules the event now.
E schedule (**double**, SimEventAction) Schedules the action at given time with default event name.
E scheduleNow (SimEventAction, String) Schedules the action now with given event name.
E scheduleNow (SimEventAction) Schedules the action now with default event name.

- The `Java` package named `jsimulation` is a library for (single-threaded) discrete-event simulation.
- The `Java` package named `jqueues` is a library for (single-threaded) discrete-event simulation of queueing systems. The library depends on `jsimulation`.
- In order to perform discrete event simulations, an event list is needed, on which events can be scheduled. The event list maintains an ordering of the events it contains in non-decreasing simulation time. Typically, a single instance of an event list is used throughout the entire simulation study. The corresponding (abstract) types for event lists and events are defined in `jsimulation`, and named `SimEventList` and `SimEvent`, respectively. This package also provides a reasonable implementation for a `SimEventList` named `DefaultSimEventList`.
- On a `SimEventList`, all scheduled `SimEvent`s are unique; you cannot schedule a `SimEvent` more than once on a single `SimEventList`. Typically, `SimEvent`s are created and scheduled through various utility methods.
- The time at which a `SimEvent` is scheduled, is kept on the `SimEvent` itself, and available though the `SimEvent.getTime` method. Once scheduled, you cannot change the time of a `SimEvent`. You can, however, reschedule it at a different time through the `SimEventList.reschedule` method.
- It is perfectly legel if multiple `SimEvent`s are scheduled at the same time. On a `DefaultSimEventList`, they are processed in random order.
- With each `SimEvent`, an action is associated that determines what to do when the event is processed by the event list. The generic type in `jsimulation` is `SimEventAction`. Unlike

`SimEvents`, `SimAction` need not be unique on the event list, and can be shared among different events.

- Once sufficient events have been scheduled, a simulation experiment starts by running or processing the event list. In `jsimulation`, you can run the `SimEventList` until it is exhausted of events through the `SimEventList.run` method, until it has reached a specific simulation time through the `SimEventList.runUntil` method, or on an event-by-event basis through `SimEventList.runSingleStep`.
- A `SimEventList` keeps a notion of simulation time. It is available through `SimEventList` ↪ `.getTime`. While running, this is always the scheduled time of the current event being processed. When not, it is always smaller than or equal to the time on the first scheduled `SimEvent`.
- You cannot schedule (at the risk of an `Exception`) a `SimEvent` with time strictly smaller than the current simulation time on the `SimEventList`.
- Event may be scheduled simultaneously, in which case their order of processing is *random*.
- Events may be scheduled at $t = -\infty$ and $t = +\infty$.
- The `SimEventList.reset` and `SimEventList.reset (double)` methods reset the event list, meaning all scheduled `SimEvents` are removed from the list, and the time on the event list is set to its default time (first method) or given time. The typical use case of these methods is running the simulation again (for instance, for variance-reduction purposes). You cannot invoke either method while the event list is being processed, at the risk of an `Exception`.

Chapter 5

Entities

In the previous chapter we introduced the core concepts of `jsimulation`; in the present chapter we delve into `jqueues`, and explain it center of gravity, viz. *(simulation) entities*. Other fundamental concepts in `jqueues` like *queues* and *jobs*, both of which are actually specific manifestations of entities, are described in subsequent chapters, as well as specific implementations of *listeners*.

Even though this chapter is quite abstract and deliberately kept compact, its contents are crucial for understanding core simulation concepts like *queues* and *jobs* in later chapters. Once you understand entities properly, we can describe queues and jobs in a highly compact, precise and almost mathematical, form. We therefore strongly recommend to take the time to go through this chapter before "jumping into" the admittedly more interesting later chapters on queues on jobs, and to return to this chapter if things are unclear later on.

5.1 Anatomy of a Simulation in jqueues

Our starting point for a detailed description of the `jqueues` package, is its vision on a *simulation run*; a concept already described for `jsimulation` in Chapter 4. We recall that in a discrete-event simulation, the *events* on an *event list* are processed one at a time in order of non-decreasing *event time*. The processing of an event involves the invocation of the event's *action*, which in turn may result in the scheduling of new events (in the future) on the event list. A simulation ends when there are no more events scheduled on the event list[1].

In `jqueues`, we (have to) narrow this view. In particular, we assume that in a discrete-event simulation (run), we are primarily interested in a specific subset of objects affected by events. In `jqueues`, these objects are called *simulation entities* and are characterized by their implementation of the `SimEntity` interface. There may be many more `Java` objects that are affected by the scheduled events, for instance, for logging, reporting or presentation purposes, or for statistical analysis, or for workload generation, but our *main interest from the problem domain is in the (modeled) entities*.

The crux is that a `SimEntity` has a well-defined *simulation state* that can change *only* as a result of processing events. However, these events cannot arbitrarily manipulate the state a `SimEntity`; they can only do so by invoking *entity operations*. An operation is a well-defined method, or set of methods, on an entity that (potentially) changes its state. In `jqueues`, the type use for operations is `SimOperation`, but you will hardly need this in practical simulations. Each entity type comes with its specific *minimal* set of admissible operations. For instance, every queue and job, both of which are entities, must support the `Arrival` operation.

An operation on a `SimEntity` is either *external* or *internal*. The former can be scheduled

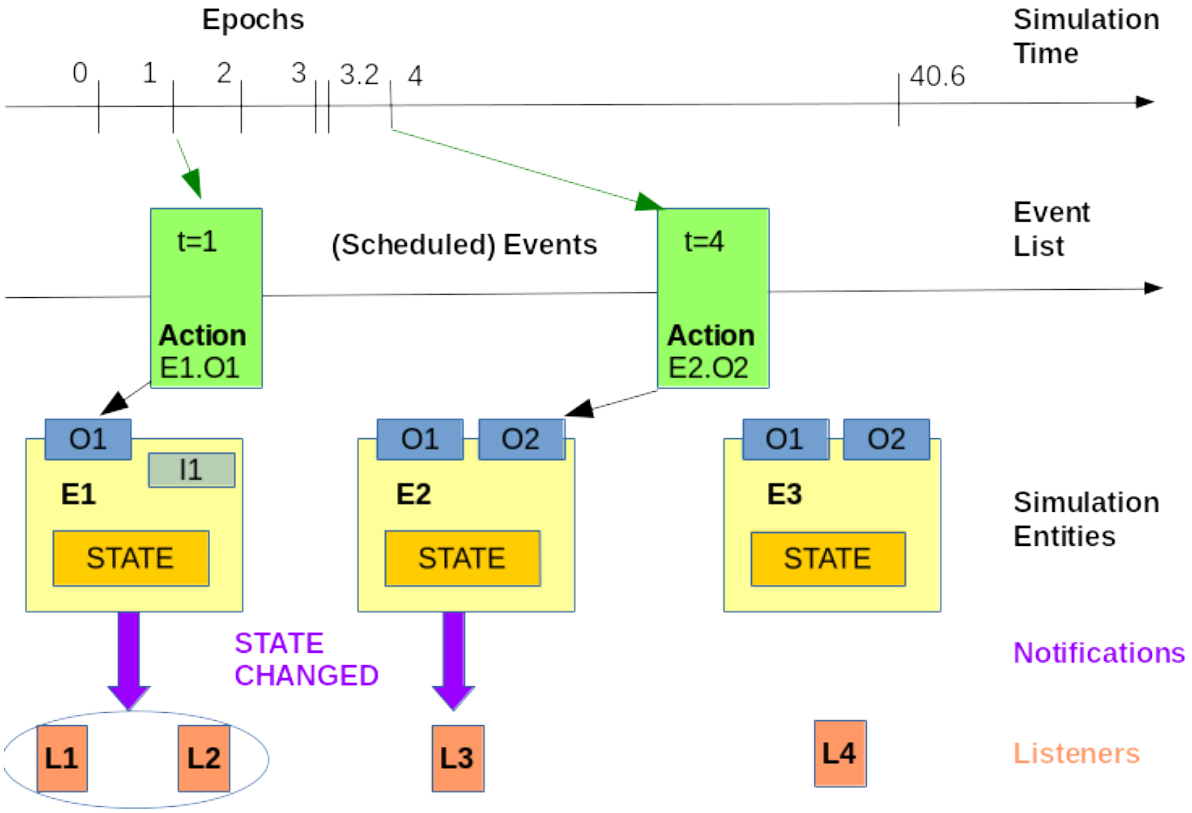
[1] Or until some other criterion is met; see Chapter 4.

by the end-user, whereas the latter can only be scheduled by the entity itself. For instance, for queues, `Arrival` is an external operation, whereas `Departure` is internal.

All entities support the notion of a *reset state*, which is the state[2] they attain upon creation, or after a so-called *entity reset*. The reset state is well defined for each entity type; for queues, for instance, the reset state requirements mandate that the queue is empty. Performing a reset on an entity is a feature typically needed when running a simulation multiple times with the same set of queues (or even jobs), for instance because a certain accuracy has to be achieved (often with *variance-reduction* techniques like *replication*). Because the semantics of an entity reset are rather complicated, and many simulation studies do not need it (because they simply perform a "single-run"), its detailed discussion is deferred. Nonetheless, it is important to realize that every entity supports a well-defined reset operation.

Finally, simulation entities are required to maintain a set of *listeners* interested in state changes of the entity. So, whenever the state changes of an entity, all of its registered listeners are informed.

Figure 5.1: The anatomy of a simulation in `JQueues`.



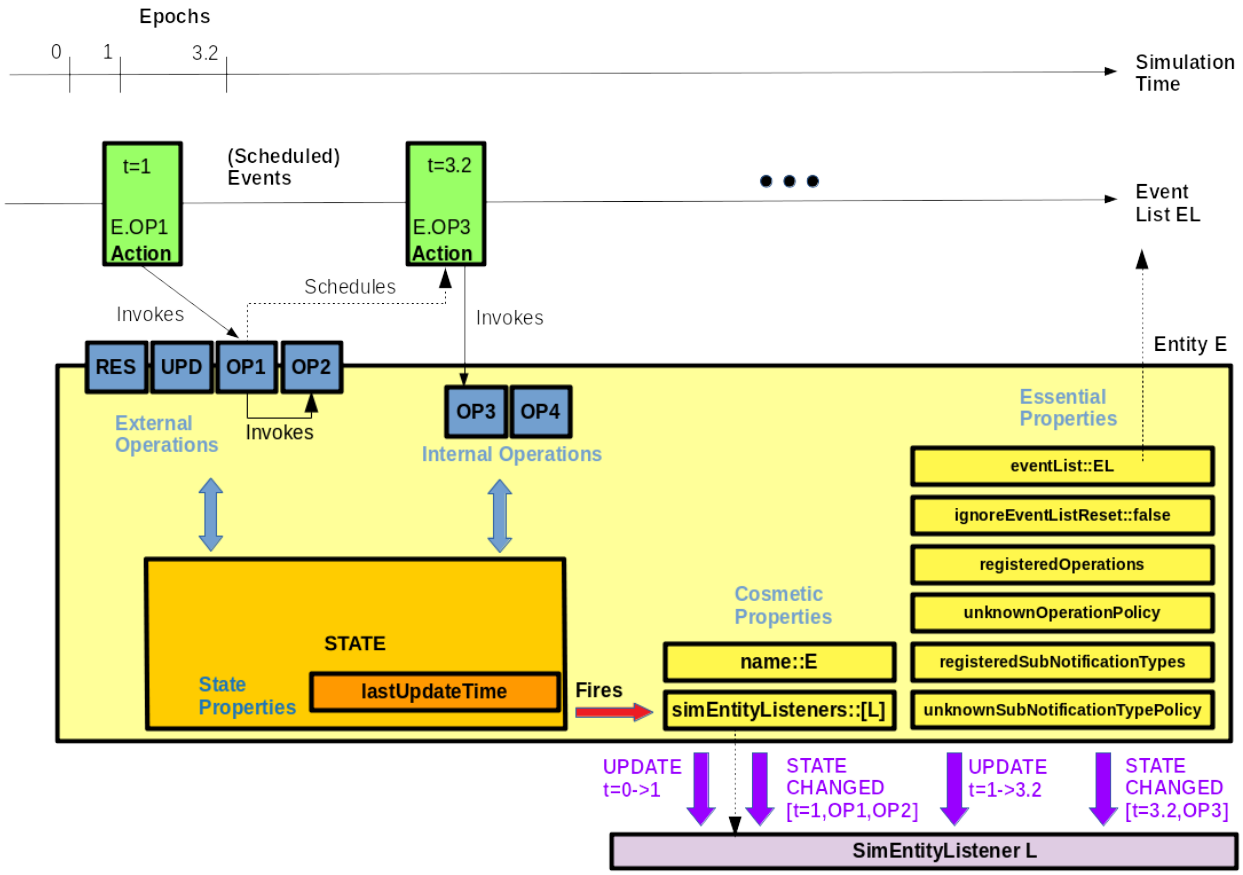
In Figure 5.1, we attempt to explain the `jqueues` concepts and features described thus far. In the top two rows we show the by now hopefully familiar concepts of *epochs*, *simulation time*, *events*, and *actions*. The event list contains two events, one scheduled at simulation time $t = 1$ and one at $t = 4$. The former's action is to invoke operation `O1` on entity `E1`, whereas the latter invokes `O2` on `E2`. After the effect of invocation of `E1.O1`, while the event list is being processed, entity `E1` notifies its registered listeners `L1` and `L2` through a `STATE_CHANGED` notification. Similarly, `E2` notifies `L3` after completing its `O2` invocation from the event list.

[2]In the sequel, whenever we refer to "the state" of an entity, we always mean its "simulation state".

5.2 The SimEntity Model in More Detail

In the present section, we refine the `SimEntity` model introduced in the previous section. In Figure 5.2, we illustrate the detailed model of a `SimEntity`. We will use this figure in our explanations of the more advanced concepts in the next sections.

Figure 5.2: The detailed model of a `SimEntity`.



5.2.1 Top-Level and Chained Operation Invocations

When a `SimEvent` (or, more precisely, its `SimEventAction`) calls a method on an entity corresponding to one of its operations this is called a *top-level operation invocation*. It is perfectly legal for an entity implementation to invoke *other* operations on it while performing an operation, i.e., still within the context of the event that triggered the top-level operation invocation, and such invocations are called *chained operation invocations*.

Why is this distinction important? Well, the golden rule in `jqueues` is that *only state changes due to top-level operation invocations on an entity must be atomically reported to that entity's listeners, and only (obviously) after the operation has completed.* It sounds complicated, but we think it makes more sense than to report each individual operation invocation, probably exposing an inconsistent state of the entity.

In Figure 5.2, at $t = 1$, `E.OP1` is invoked from the event list, so this invocation is a top-level one. However, *while* `E.OP1` *is being processed*, the entity invokes `E.OP2`, which is therefore a chained invocation. It is crucial to understand that while `E.OP1` also schedules the invocation of `E.OP3` at $t = 3.2$, the latter when processed will be a top-level invocation on its own. Referring to the lower right part of the figure, a listener thus receives a mandatory `UPDATE` notification at $t = 1$, indicating a non-trivial progress of time (from $t = 0$ to $t = 1$; we silently assumed that

the simulation starts at $t = 0$) without state changes. It then receives a single `STATE_CHANGED` notification from the `E.O1` invocation at $t = 1$, but notification holds both `E.O1` and `E.O2` as sub-notifications, because `E.O2` was invoked from within the context of `E.O1`. Their joint effect on the state is thus described in a single atomic notification.

5.2.2 Top-Level Notifications and Sub-Notifications

In the previous section we learned that entities are obliged to report state changes atomically. On order to avoid potential confusion, we need to properly introduce some notification-related terminology.

A *top-level notification* is the invocation of a method on the registered listeners of an entity. There exist only the following three types of top-level notifications:

- `RESET`: The entity has been put into its reset state, either because its underlying `SimEventList` ↪ was reset, or because its `Reset` operation was invoked. A `RESET` notification is equipped with the *old* time, i.e., the time *before* the reset, and the *reset time*, i.e., the *new* time on the entity[3]. It is important to realize that a `RESET` notification is the only one allowed to "set back" the time on the entity, and that it is only issued as a result of the invocation of its `Reset` operation, typically induced from resetting the event list. Hence, a `RESET` notification is *not* issued upon construction of the entity, *not* upon its attachment to a `SimEventList`.

- `UPDATE`: The entity is *about to* change its state because an operation on it was invoked, and it exposes *its old state* and the time at which the update occurs. The `UPDATE` notification primarily targets statistics-gathering listeners that need to know the interval length during which the entity's state remained constant.

- `STATE_CHANGED`: The entity has been subject to the external invocation of one of its operations, and it exposes its *new* state and the time at which and the manner in which the new state was reached. The description of the transformation of the old state into the new one is done through a sequence of *sub-notifications* described below. A `STATE_CHANGED` notification is not issued when the entity has been reset.

Note that there is no dedicated `Java` type for top-level notifications, simply because at the present time, we see no purpose in extending the current three-sized set of top-level notifications.

A *sub-notification* is a precise description of an aspect of an entity's state transformation. For the purpose of this description, it is always part of a top-level `STATE_CHANGED` notification. So, in itself, a sub-notification is incomplete. However, it unambiguously reports a state transformation, and the state's a priori and a posteriori conditions. So given an entity's state (that is valid) and a sub-notification, one can always predict the new state of the entity. Often, it just takes multiple sub-notifications in sequence to describe a `STATE_CHANGED` notification, which explains their existence.

A sub-notification is a tuple of its *type*, an object implementing `SimEntitySubNotificationType` ↪ , and its arguments. The actual type of a sub-notification defines its required arguments and their structure. By default, a `SimEntity` can only emit sub-notification types in a `STATE_CHANGED` notification that have been *registered* at that entity. The process of registering sub-notification types is taken care of in the constructors of the various `SimEntity` sub-types and concrete implementations.

[3] See the description of the `lastUpdateTime` further in this chapter.

5.2.3 Properties

In `jqueues`, we adopt the notion of object *properties* from `JavaBeans`, and we silently assume that the reader is familiar with the basic concepts of `JavaBeans`. We follow the property-naming conventions, and in `jqueues`, we only use property names starting with a lower-case letter.

For any `SimEntity`, we classify its properties as follows[4]:

- **State Properties** are part of the entity's simulation state; their values can only change as a result of the invocation of registered operations on the entity. You should never (attempt to) set them directly from user code. For a queue, for instance, the set of currently visiting jobs is a state property.

- **Essential Properties** are *not* part of the entity's simulation state; they do, however, affect the functionality of an entity's operations, *and* the entity assumes that their values *remain constant while running a simulation*. Hence, their values can only be changed under certain conditions, and certainly *not* (by whatever means) *while running a simulation*. The buffer size of `FCFS_B`, a `FCFS` queueing system with finite buffer space, is an example of an essential property, because its value decides whether or not an arriving job is *dropped* in the presence of jobs present. In other words, it affects among others the `Arrival` operation.

- **Cosmetic Properties** are *not* part of the entity's simulation state, but unlike essential properties, their values can be changed at any time without affecting the simulation state of the entity or the functionality of any of its operations. Their values may even be changed from an event. The name of an entity is a cosmetic property; its value, by contract, never affect the state of the entity nor the functionality of its operations.

Note that many sub-types of `SimEntity` put even stronger restrictions on setting an essential property. In many cases, you can only set it *upon construction*, or only *immediately after a* `Reset` operation.

5.3 Mandatory Properties of a SimEntity

In Table 5.1, we list the mandatory properties of a `SimEntity`, classifying them into state, essential and cosmetic properties introduced previously.

The only mandatory state property is `lastUpdateTime` holding the (a posteriori) simulation time of the last (invocation of the) `Update` or `Reset` operation, whichever occurred last. Note that upon construction, a `SimEntity` always enters its reset state, which sets the initial value of `lastUpdateTime`. This implicit `Reset`, however, is *not* notified to listeners because there cannot be any.

Among the essential properties are `registeredOperations` and `registeredSubNotificationTypes`, reporting the registered operations and notification sub-types, respectively. Typically, implementations will return unmodifiable views on the underlying collections. There are also essential properties that govern a `SimEntity`'s behavior when an unregistered operation on it is invoked, or when its implementation asks for the emission of a sub-notification it does not know. Both features are of little practical use, except perhaps for testing purposes. Last but not least, the `simEventList` property holds the event

[4]This classification is `jqueues`-specific; it is not part of `JavaBeans`.

[5]In fact, `SimEntitySimpleEventType.Member`.

[6]`UnknownSubNotificationTypePolicy`.

Table 5.1: Mandatory properties of a `SimEntity`.

Name	Type	Default/Reset
SimEntity		
State Properties		
`lastUpdateTime`	`double`	From `eventList`; $-\infty$ if absent.
Essential Properties		
`eventList`	`SimEventList`	From constructor; may be `null`.
`ignoreEventListReset`	`boolean`	`false`.
`registeredOperations`	`Set<SimEntityOperation>`	Sub-type-dependent; set in constructor(s).
`unknownOperationPolicy`	`UnknownOperationPolicy`	`ERROR`; settable by user.
`registeredNotificationTypes` *deprecated*	`Set<Member>`[5]	Sub-type-dependent; set in constructor(s).
`registeredSubNotificationTypes` *since r5.3*	`Set<SubNotificationType>`	Sub-type-dependent; set in constructor(s).
`unknownNotificationPolicy` *deprecated*	`UnknownNotificationPolicy`	`ERROR`; settable by user.
`unknownSubNotificationTypePolicy` *since r5.3*	`UnknownSubNot...Policy`[6]	`ERROR`; settable by user.
Cosmetic Properties		
`name`	`String`	`null`.
`simEntityListeners`	`List<SimEntityListener>`	Empty `List`.

list to which the entity is attached. Typically, the event list is passed to the constructor of concrete `SimEntity` implementations, and setting it properly is actually mandatory for certain sub-types like queues (`SimQueue`). However, *if* a `SimEntity` is attached to a `SimEventList`, the former will register at the latter and automatically perform a `Reset` upon a reset of the event list. This behavior is controlled through the `ignoreEventListReset` property, but this should *never* be changed by user code; it is strictly meant for implementation use.

The two mandatory cosmetic properties on a `SimEntity` are its registered listeners (`simEntityListeners`), which will be described in more detail in the next section, and its `name`. Despite being a "cosmetic" property, setting the `name` of a `SimEntity` is often quite important. Typically, if you do not set it, implementations of `SimEntity` will use a *type-specific* `String` for the `toString` method; otherwise, they will supply the value of the `name` property. So, if you are studying a single queue like `FCFS`, there is really no need to set its name; its default `toString` representation will be `"FCFS"`, which is clear enough. If, however, you use multiple queues of the same type, or if distinct individual identification is required of the *jobs* (`SimJob`) the queue(s) is/are subject to, it is highly recommended to set the `name` property accordingly. (Another, implementation-driven argument for this approach is that end users can set an entity's name *even if the implementation is final*. On such implementations, one clearly cannot override `toString`.)

5.4 Registering and Unregistering Listeners

As stated previously, a `SimEntity` must always report state changes atomically to each registered listener. In `jqueues`, such listeners have type `SimEntityListener`, and they can be registered and unregistered at any time on a `SimEntity` through the `registerSimEntityListener` and `unregisterSimEntityListener` methods, respectively. The set of currently registered listeners of an entity is available through its `simEntityListeners` property, which is a cosmetic property and thus *not* part of the entity's state.

5.5 Mandatory Operations on a SimEntity

Every `SimEntity` *must* support the following operations, all of which are *external*:

- `Reset`: This operation puts the entity into its reset state, and sets its `lastUpdateTime` property to the time argument provided. It is the *only* operation allowed to "set back" the time on the entity. The `RESET` operation is not meant to be invoked directly from user code. If the `SimEntity` is attached to an event list, through its `simEventList` property, it will reset itself automatically when the attached event list resets. A `SimEntity` will issue a `UPDATE` notification first when its `Reset` operation is (explicitly) invoked, and after completion, it will notify its listeners with a `RESET` notification. Upon construction, a `SimEntity` always enters its reset state, possibly overriding state-property values (directly or indirectly) from the constructor arguments. This is an implicit invocation of `Reset` which is not reported to listeners.

- `Update`: This operation sets the `lastUpdateTime` property on the entity from the time argument supplied, and notifies listeners through the `UPDATE` notification, providing both a priori and a posteriori simulation time. It *never* modifies state properties other than `lastUpdateTime`.

- `DoOperation`: This "meta operation" invokes the operation provided as argument. The operation has to be registered at the `SimEntity`; if not the behavior of the entity is

controlled by the value of its `unknownOperationPolicy` property. The operation-specific arguments have to be provided as well.

5.6 Summary

In this chapter, we introduced the core concept of `jqueues`, viz., `SimEntity`s. Below, we summarize the contents of this chapter:

- In `jqueues`, we are primarily interested in the behavior in simulation time of objects named *(simulation) entities*.
- Simulation entities model objects from the real-world problem domain; they are represented as `SimEntity Java` objects. Queues and jobs are typically instantiations of entities.
- A `SimEntity` has a well-defined *simulation state*, which is a subset of the `Java` object state. The simulation state can only change as a result of the invocation of registered `SimOperation`s on the entity.
- The invocation of an operation on a `SimEntity` from the event list is called a *top-level operation invocation*. Such invocation may induce the invocation of other operations on the `SimEntity`; these are called *chained invocations*.
- A `SimEntity` processes and reports its top-level operation invocations *atomically*. The reports of state changes are through `STATE_CHANGED` notifications, which consists of a sequence of *sub-notifications* each describing a specific, well-defined, transformation of the entity's state.
- Like operations, sub-notifications need to be of a known type, well documented, and registered at the `SimEntity`.
- A `SimEntity` exposes its `Object` state through `JavaBeans` properties. A property can be a *state* property, representing an aspect of the entity's simulation state, a *essential* property, having dominant effect on the behavior of operations, or a *cosmetic* property, which does *not* have that effect.
- Each `SimEntity` must support the `Reset`, `Update` and `DoOperation` operations.
- Each `SimEntity` must support the `RESET`, `UPDATE` and `STATE_CHANGED` top-level notifications.

Chapter 6

Jobs and Queues

In the previous chapter, we introduced the concept of a *simulation entity*, with type `SimEntity`, as being the predominantly interesting object in a discrete-event simulation. In the current chapter, we narrow this down to the use case of *queueing systems*, and we introduce *jobs*, with type `SimJob`, and *queues*, with type `SimQueue`, as the most interesting manifestations (specializations if you will) of a `SimEntity`.

Our strategy in describing both `SimJob` and `SimQueue` in this chapter is to introduce their structure (properties) and behavior (operations and notifications) in a concise manner, using the foundation laid in Section 5, without elaborating into examples. We believe that both concepts are best understood when presented in a compact yet complete form. Moreover, we cannot introduce examples without concrete implementations of `SimJob` and `SimQueue`.

6.1 Visits, Arrivals and Departures

The central notion in `jqueues` is that a queueing-system simulation (run) consist of a sequence of *visits* of entities named *jobs*, `SimJob`s, to others named *queues*, `SimQueue`s. A `SimJob` can visit *at most one* `SimQueue` at a time but there can be an arbitrary amount of (simulation) time between successive queue visits. A `SimJob` may revisit the same `SimQueue` arbitrarily many times, provided that, obviously, these visits do not mutually overlap in time.

The `SimQueue` currently visited by a `SimJob` —if any— is maintained in the job's `queue` state property. If the job is not currently visiting a queue, the property value is `null`. Likewise, the set of `SimJob`s currently visiting a particular `SimQueue` is maintained in that queue's `jobs` state property. We refer to the "area" holding visiting jobs in a `SimQueue` as its *visit area*. If no job is currently visiting the queue, the latter's `jobs` property value is the empty set.

Since both `SimJob.queue` and `SimQueue.jobs` are state properties, they are under well-defined control of the operations on the respective entities. The most important operations are the *arrival* of a job at a queue. and the *departure* of a job at a queue. Both will sound familiar to readers with some background in queueing theory.

In `jqueues` release 5, the only way to initiate[1] a visit is through invocation of the `Arrive` operation, which is external (and thus, user-schedule-able) on both `SimJob` and `SimQueue`. It is a *binary operation*; you can invoke it on any of the two entities (job or queue), providing the other entity as argument. The corresponding sub-notification type is `ARRIVAL`.

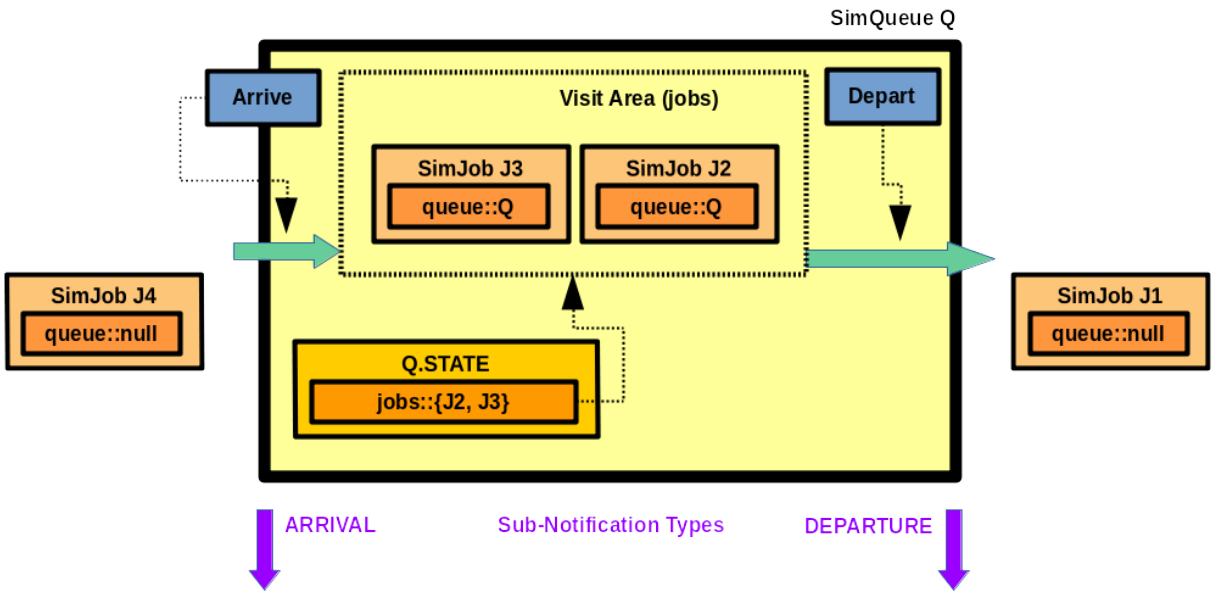
If the visit ends "normally", i.e., the visiting `SimJob` has obtained its required service or some other requirement is met, the `SimQueue` invokes its internal `Depart` operation, with corresponding sub-notification type `DEPARTURE`. Both the `ARRIVAL` and `DEPARTURE` sub-notifications provide the `SimJob` and `SimQueue` as argument to (entity) listeners in the `STATE_CHANGED`

[1] We carefully avoid using 'start' in this context, in order to avoid confusion later on.

notification, see Section 5.2.2.

In Figure 6.1, we show the partial state of a `SimQueue` and `SimJobs` that visit it. (In subsequent sections, we will extend this figure until it finally captures all state properties and operations on a `SimQueue`.)

Figure 6.1: Arrivals and departures at a `SimQueue`.



In the figure, jobs `J2` and `J3` are currently visiting `SimQueue Q`; job `J1` just departed from `Q` and `J4` is about to arrive at `Q`. On `Q`, the `jobs` property equals `{J2, J3}`; the set holds the jobs in its visit area. Both `J2` and `J3` have their `queue` property set to `Q`; on the other jobs it is set to `null`. The latter is obviously a prerequisite for jobs to be able to arrive at any queue.

The concept of a visit of a job to a queue is fundamental in queueing theory, and we believe we have captured its most generic aspects in `SimJob` and `SimQueue`: a job arrives at a queue for a visit through whatever means; we are not concerned with the reason of the visit, nor with the nature of the "service" that the queue will give to the job. All we care about is the fact that the job arrives at a certain time, and, at the discretion of the queue at hand, departs from it at some later time, or even immediately. The only real constraint we have set is that a job can visit at most one queue at a time, but this fits, to the author's knowledge, most use cases of queueing systems.

We want to stress that the only way in which to initiate a visit of a job to queue is through scheduling an `Arrive` operation on the event list; neither a job nor a queue will ever schedule that event by itself. This means that arrivals can never be used by a `SimQueue` implementation as a means to obtain its specification. But if `Arrive` is invoked on a queue that already holds the job provided, or if the job is already visiting another queue, an `Exception` is thrown.

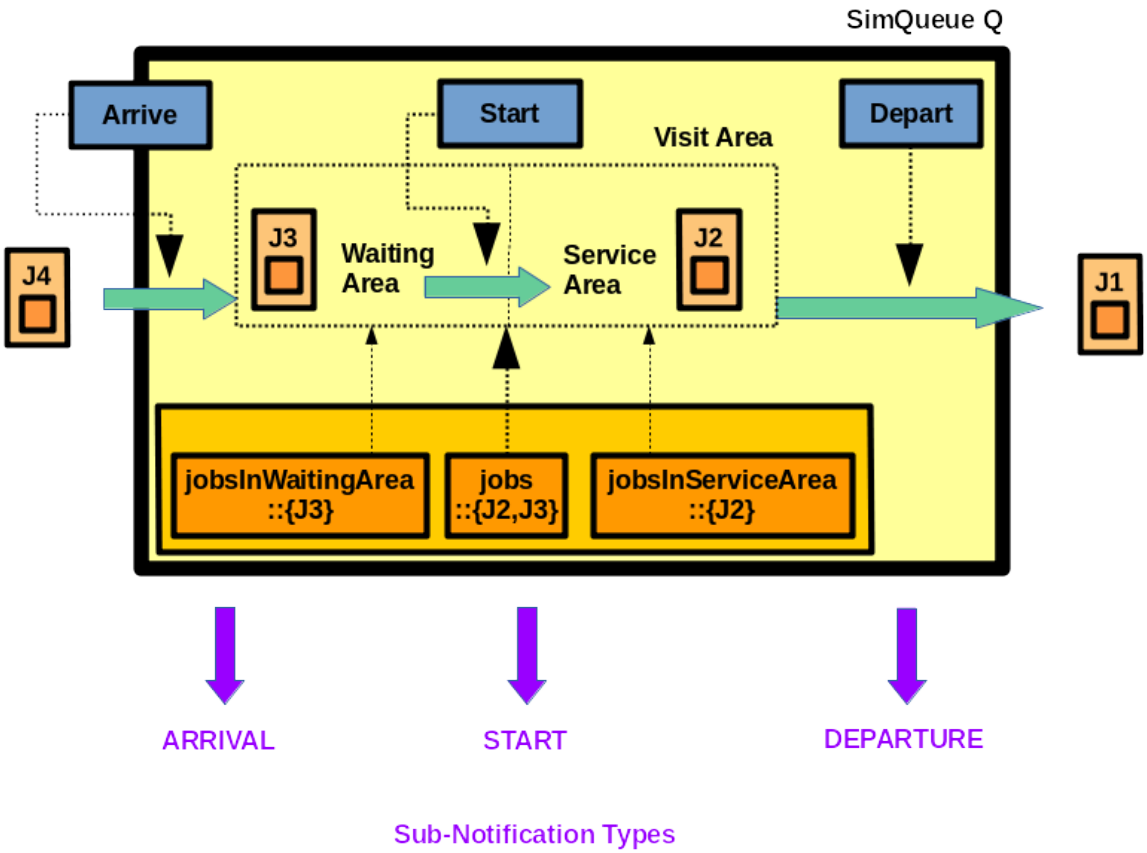
Unlike the case with arrivals, a departure is *not* the only means by which a visit can end. In sections further on, we introduce *dropping*, *revoking* and *auto-revoking* a job as alternative means in which a queue visit can end. We then briefly describe the case in which a visit *does not end at all*, by explaining so-called *sticky jobs*. However, before doing that we need to elaborate on the *partitioning* of the visits area into two separate areas.

6.2 The Waiting and Service Areas and the Start Operation

In `jqueues`, the visit area of every `SimQueue` is partitioned into a *waiting area* and a *service area*; visiting jobs are always present in precisely one of them. Upon arrival, jobs always enter the waiting area. At the discretion of the `SimQueue`, the may move from the waiting area to the service area through the internal `START` operation. An important and perhaps non-intuitive restriction is that *jobs cannot move back from the service area into the waiting area*. Note that jobs may depart from either area. A `SimQueue` maintains the visiting jobs present in the waiting and service areas in its `jobsInWaitingArea` and `jobsInServiceArea` state properties, respectively. Both properties are typed as `Sets` of `SimJobs`.

In figure 6.2, we illustrate the partitioning of the visit area into a waiting area and a service area.

Figure 6.2: The waiting and service areas of a `SimQueue` and its `Start` operation.



In the figure, jobs `J3` is in the waiting area, as mandated by the queue's `jobInWaitingArea` (state) property. Likewise, job `J2` is in the queue's service area, in other words, it has *started*, and it is in the queue's `jobsInServiceArea` set.

The model for queues in terms of waiting and service area is, admittedly, somewhat deviant from models in literature. The main point is that we make *no* assumptions whatsoever on the structure of the waiting and service areas. But for most known queueing systems, the waiting area is simply a queue holding waiting jobs, often in FIFO (First-In First-Out) order, and the service area consists of one of more servers serving jobs until completion. In (classical)

processor-sharing queues, there is virtually no waiting area, as jobs enter the service area immediately.

The most obvious complications with this partitioning compared to "classical approaches and viewpoints" is that since jobs cannot move back from the service area into the waiting area, one has to let go of the intuitive notion that jobs in the service area area actually *being served*. Although true for many queueing systems, it is false for systems like Preemptive/Resume Last-Come First-Served (`P_LCFS`), and many other preemptive queueing systems. In `P_LCFS`, whenever a job in the service area is preempted in favor of a new arrival, the former stays in the service area, yet it is not *served* (at least, not for a while). Another complication, actually induced by `jqueues` itself, is that in order to start a job, a queue needs at least one so-called *server-access credit*, but by default this is always the case; it is explained in more detail in Section 6.8.

6.3 The Drop Operation

In the previous sections we noted that the "usual" means in which a visit ends (after the visiting job has acquired all it needs from the queue) is through the `Depart` operation. However, a `SimQueue` may decide that these requirements cannot or can no longer be met due to queue-specific *constraints*, and that the job needs to end its visit *without having acquired its purpose*. For instance, the queue may not or no longer have enough buffer space to allow the job's visit, or it may have limits on the allowed duration of a visit. In such cases, a `SimQueue` *drops* a `SimJob` through invocation of its *internal* `Drop` operation. A job drop may happen while the job is in the waiting area or while it is in the service area. It may even happen immediately upon arrival of a job. When a job is dropped at a queue, both will issue a `DROP` sub-notification.

In Figure 6.3, we show the modified simple model of the state of a `SimQueue`s and `SimJob`s of Figure 6.2, now including the `Drop` operation and the corresponding sub-notification type.

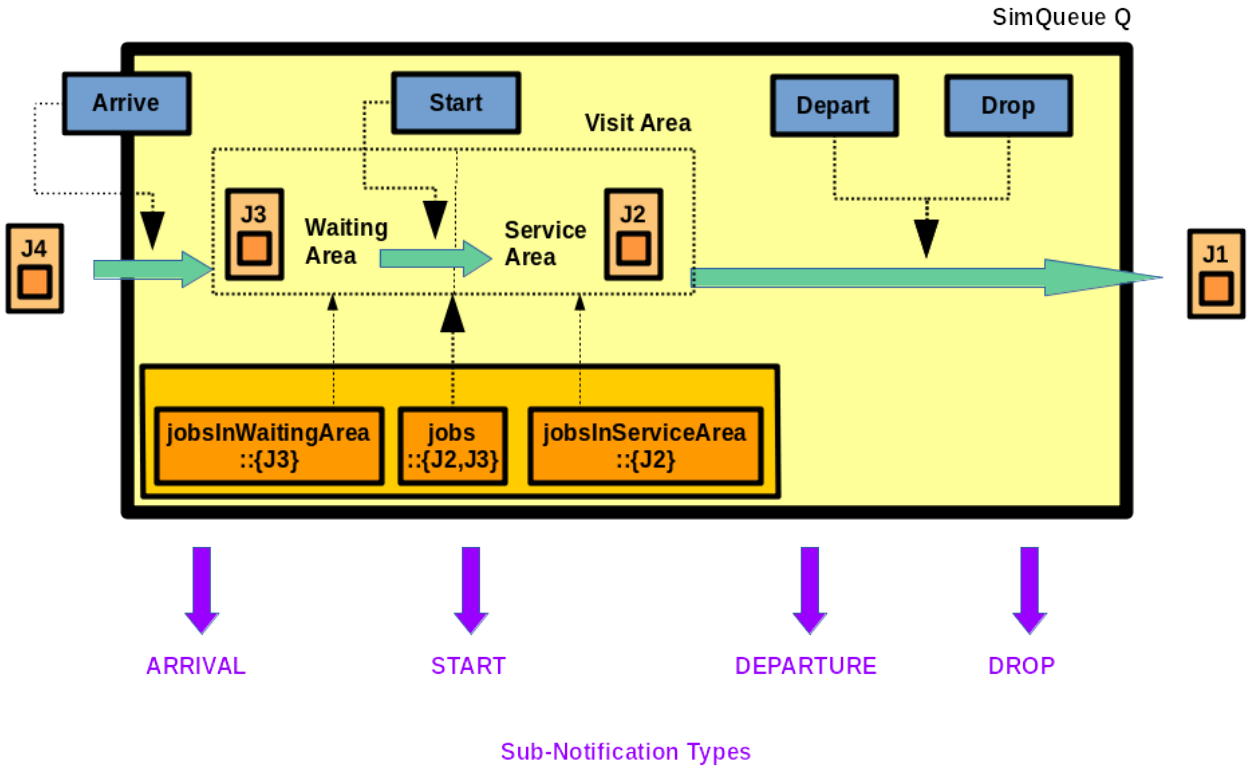
In the modified version of the figure, job `J1` may have *departed* or have been *dropped* from `Q`; the effect of either is the same on `J1`, i.e., a job has no notion of having departed or having been dropped from its latest visited `SimQueue`.

It is important to realize that the invocation of the `Drop` operation is *always initiated by the queue during a visit*; you cannot enforce it through an event on the event list, nor (directly) from an operation on the `SimJob` in question. Since by default, generic `SimQueue`s have no reason to drop jobs, specific implementations that do must precisely specify (1) the conditions under which jobs are dropped and (2) the selection of which job is dropped once that condition is met.

The mechanism of dropping a job is present in many classical or more advanced queueing models. The most common reason for dropping jobs is *lack of buffer space*: A job arrives at a queueing system while all buffer space in that system has already been used for other visiting jobs. The queueing system then resorts to dropping a single (in most cases) job. Note that this may not be the arriving job! The classical examples of systems that employ dropping jobs for lack of buffer space are the `FCFS_B` (First-Come First-Served with Finite Buffer Size `B`) and `LCFS_B` (Last-Come First-Served with Finite Buffer Size `B`) queues. The former will drop an arriving job that finds a "full queue", in this case, that finds all available places in the waiting area occupied. The latter, however, will drop the job in the waiting area with the earliest arrival time. If no such job is present, i.e., the system has zero buffer size, the arriving job is dropped instead.

A second reason for dropping jobs is a given limit on the job's *waiting time*. In this case, a job is dropped from the waiting area once its time spent there exceeds some given threshold. Many models with *impatient customers* fit this case. Obvious variants put limits onto the *time spent in the service area* or onto the *sojourn time* of a job. Combinations are also possible. In

Figure 6.3: Conceptual illustration of arrivals, departures, start and drops at a `SimQueue`.



`jqueues`, you can actually turn *any* `SimQueue` into one with impatient customers through the `EncTL` *composite* queueing system implementation. For more details, we refer to Section **XXX**.

6.4 The Revoke Operation

Up to now, we have introduced job departures and drops as two means by which a `SimJob` can end its visit to a `SimQueue`. Both are invoked at the discretion of the queue. The third and final way in which a visit can end is through *revocation*; the user-initiated exit of a job at a queue.

Revocations actually come in two flavors:

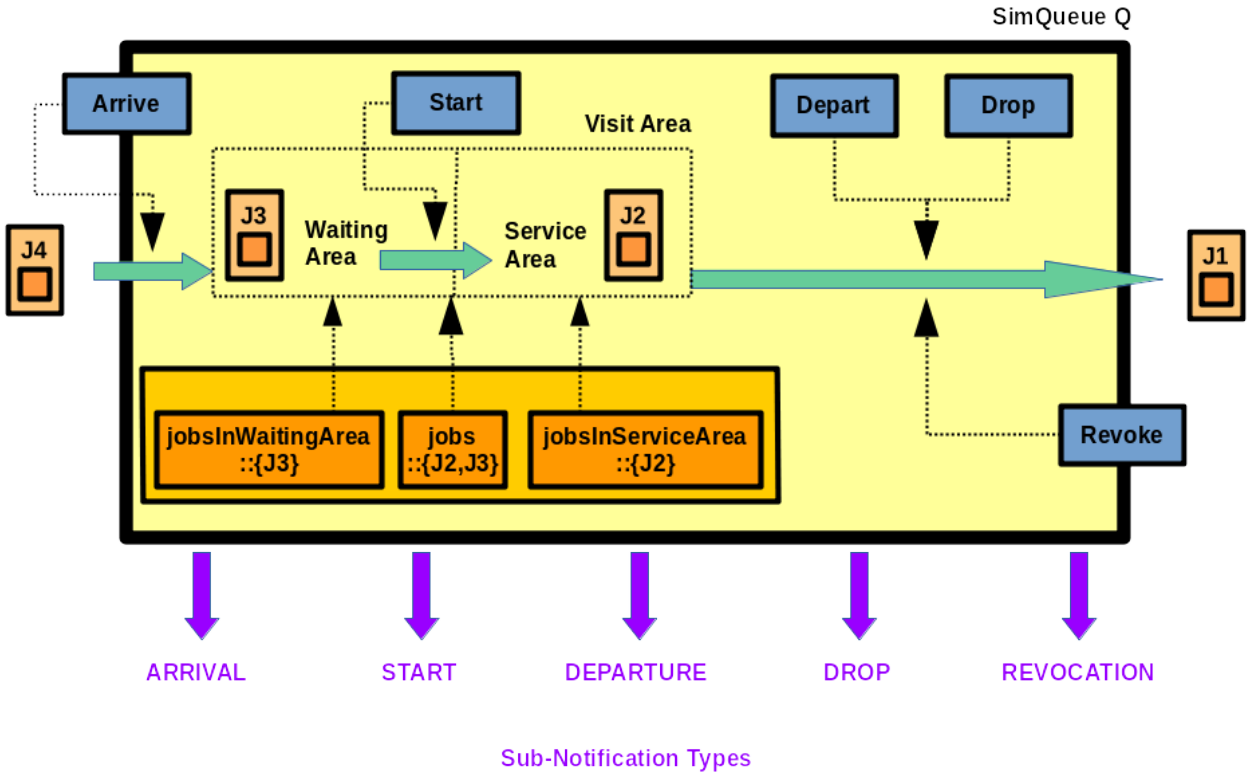
- Users can invoke the external `Revoke` operation, requesting for the forced exit of a jobs. Every `SimQueue` must support the operation.

- Users can set one or more conditions on a queue, or a combination of them, that trigger the automatic removal of the job once the condition is met. Such revocations are named *auto-revocations*; we describe them in more detail in Section 6.5.

The external `Revoke` operation *requests* the removal of a job visiting a queue. In its most basic form, the so-called *unconditional revocation*, this request cannot be denied by *any* `SimQueue` implementation. In other words, every `SimQueue` supports unconditional revocation of a job. After a job has been revoked, a `REVOCATION` sub-notification is fired.

In Figure 6.4, we show the modified simple model of the state of a `SimQueues` and `SimJobs` of Figure 6.3, now including the `Revoke` operation.

Figure 6.4: Conceptual illustration of arrivals, departures, starts, drops and revocations at a `SimQueue`.



In addition to the unconditional revocation, `SimQueue` implementations may provide variants of the `Revoke` operation that take an additional condition or combination of conditions on the state of the queue and/or job. However, an invocation of such a *conditional* revocation cannot fail if the relevant condition is met

On every `SimQueue`, the method `revoke (double, SimJob)` revokes a job unconditionally from the queue. The first argument is (as always) the simulation time of the request. If the job is present a priori, the revocation request cannot fail; every `SimQueue` implementation must honor it. In the variant method `revoke (double, SimJob, boolean)`, which is actually present on *any* `SimQueue`, the third argument indicates whether it is allowed to revoke the job from the *service area*. If the argument is `false` and the job is indeed in the service area, the request will fail in a non-fatal way: No revocation takes place and no sub-notification will be fired. If, however, the job is in the waiting area, and/or the argument is set to `true` and the job is present in either area, then, again, the request cannot fail.

6.5 The AutoRevoke Operation

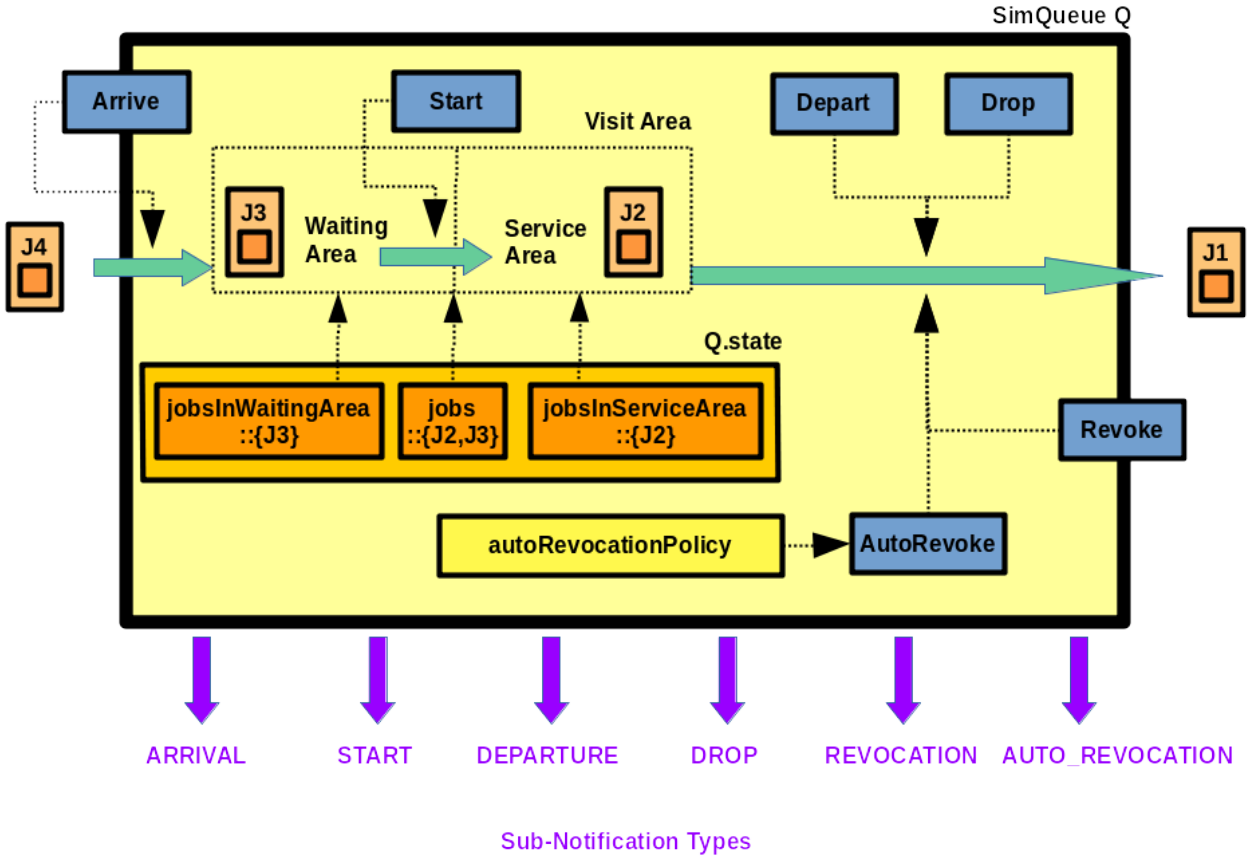
Auto-revocations are forced removals from a `SimQueue` because a user-set condition on the queue is met. The set of conditions for auto-revocation that can be set on a `SimQueue` depends on the queue's type, however, every `SimQueue` must have *any* auto-revocation condition *disabled by default*. The only auto-revocation condition every `SimQueue` *must* support, is the start of a job. (But, it must be disabled by default.) Auto-revocation is an internal operation name `AutoRevoke`; the corresponding sub-notification is `AUTO_REVOCATION`.

The condition(s) on a `SimQueue` that trigger auto-revocation is captured in the queue's `autoRevocationPolicy` property. It is an essential property, meant to be set *only* immediately

after construction or a `Reset` invocation. In Release 5 of `jqueues`, the property has `enum` type with possible values `NONE` and `UPON_START`. With value `NONE`, auto-revocation is essentially switched off. whereas with `UPON_START`, *any* job that is about to start on a `SimQueue` is automatically revoked.

In Figure 6.5, we show the modified simple model of the state of a `SimQueues` and `SimJobs` of Figure 6.4, now including the `AutoRevoke` operation and the essential `autoRevocationPolicy` property.

Figure 6.5: Conceptual illustration of arrivals, departures, starts, drops, revocations and auto-revocations at a `SimQueue`.



We do not expect many practical use cases requiring auto-revocations. Moreover, we did not add the feature with specific use cases in mind; we merely needed the feature as a part of the *generic* `SimQueue` interface in some of its specific implementations, in particular, the `CTandem2` queueing system described in Section **XXX**. If at all possible, we advise against the use of auto-revocations and manipulation of the `autoRevocationPolicy` property. Then again, the concept is available on *any* `SimQueue`, should you need it.

6.6 QoS

So far, we have introduced and described queueing systems that treat (or, distinguish, if you will) jobs based upon for instance their arrival time (`FCFS`, `LCFS` and derivatives), or their requested service time (`SJF` and `LJF`). In so-called *multiclass queueing systems*, jobs are (in addition) treated differently according to their membership to a *class* of jobs. An easy example is the queueing system associated with checking in for an flight, with multiple quueues distinguishing between business-class passengers, and poor others. Multi-class queueing systems

are often referred to as *priority systems* or *QoS (Quality of Service) systems.* In `jqueues`, we mainly use the latter terminology, viz., jobs belong to a *QoS class*, in order to avoid name clashes with the Java `class` concept.

Every `SimQueue` and `SimJob` possess two QoS-related essential properties, viz., `qosClass` ↪ and `qosValue`. On a `SimQueue`, the `qosClass` property holds the type (i.e., a `Java Class` object) the queue uses to distinguish among jobs *for reasons of QoS*. On a `SimJob`, the property value holds the `Java class` the job uses to indicate its relative priority or its membership to a specific QoS class. The actual object representing the priority of job class is held in the job's `qosValue`. This property also exists on every `SimQueue`, but its semantics are *undefined* by default, and their `qosValue` is therefore always `null`.

The anticipated use cases are that:

- In a *non-QoS scenario*, both `qosClass` and `qosValue` have their `null` default values.
- In a *QoS scenario*, the `qosClass` properties on jobs and queues must "match" and be non-`null`; the matching refers to the requirement that `qosClass` on the job must be an sub-class or implementation of `qosClass` on the queue. The `qosValue` property on the job holds the relative priority or job-class to which the job belongs; the value must be an object of type advertised by the job's `qosClass` property, or a sub-class of it. The `qosValue` on *any* queue is ignored and `null`.

Somewhat unfortunately, there is a substantial number of corner cases that deviate from the two use cases above. For instance, specific `SimQueue` implementations feature special treatment of visiting `SimJob`s with `null` or incompatible value for their `qosClass` property, or with a `null` value for their `qosValue` property. We hope to describe these corner cases in a future version of this document, but for now we refer to the `javadoc` specification for further details.

At this point, we have described *all* binary operations, internal as well as external, on `SimJob` and `SimQueue`. In the next sections, we will describe operations, properties and sub-notification types specific *only* to `SimQueue`.

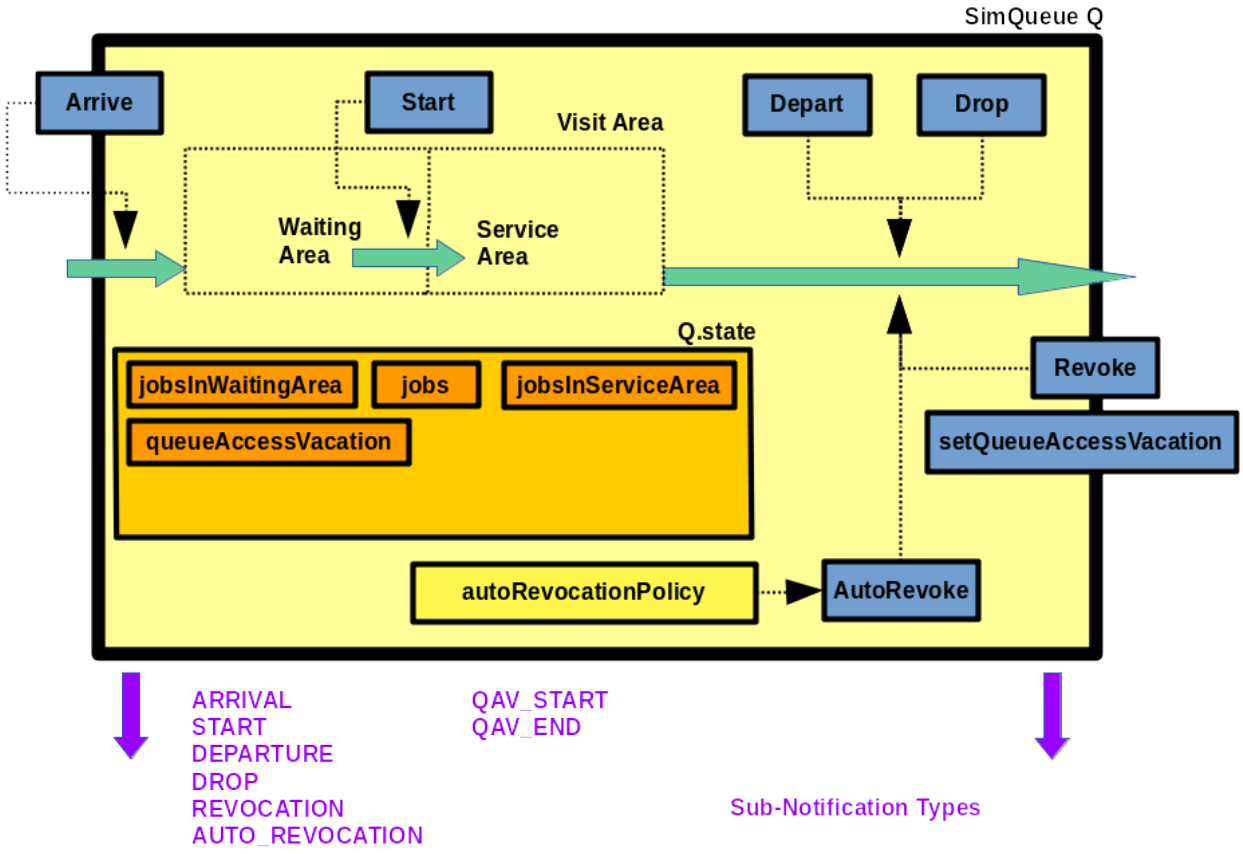
6.7 Queue-Access Vacations

In `jqueues`, every queue, in other words, every `SimQueue` implementation, *must* support the notion of so-called *queue-access vacations*. During a queue-access vacation, *all arriving jobs are dropped*, but jobs already visiting the queue are not affected. In terms of queue state, every `SimQueue` has a state property `queueAccessVacation` of type `boolean` that determines whether or not the queue is "on vacation". Starting and stopping queue-access vacations is an external operation named `SetQueueAccessVacation`, taking a `boolean` argument to indicate whether the vacation starts or ends; the corresponding sub-notification types are `QAV_START` and `QAV_END`. Be aware that these sub-notifications are *only* issued when the `queueAccessVacation` property value *actually changes*.

In Figure 6.7, we show the modified simple model of the state of a `SimQueue`s now including Queue-Access Vacations.

It is essential to note that queue-access vacations are *always* available to you as an independent means to drop arriving jobs because you think this is the right thing to do at this time. In other words, `SimQueue` implementations are *not* allowed to use the feature to get "their job done". This turns the `SetQueueAccessVacation` operation into a purely *external* one. For instance, in our previous example with `FCFS_B`, the queue *could* use queue-access vacations in order to drop jobs upon arrival if the buffer is full. But, it is not allowed to do that, and it simply never touches the `QueueAccessVacation` property.

Figure 6.6: `SimQueue` model with Queue-Access Vacations.



Scheduling the start and end of queue-access vacations on a queue is easily achieved through the utility method `SimQueueEventScheduler.scheduleQueueAccessVacation (SimQueue, double, boolean)`; the respective arguments being the queue to which the event applies, the scheduled time, and whether to start or end a queue-access vacation, respectively.

6.8 Server-Access Credits

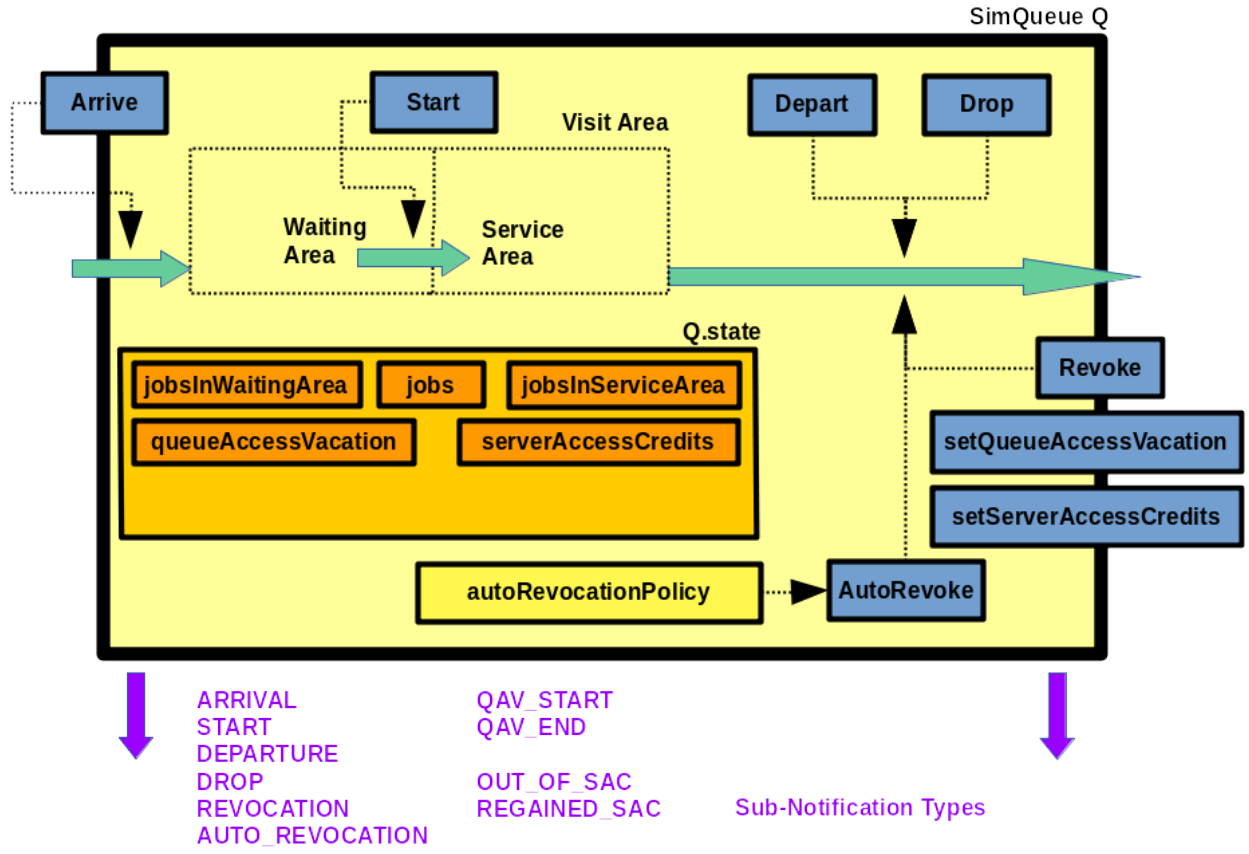
Every `SimQueue` has the `serverAccessCredits` state property of type `int`, and setting its value is an external operation named `SetServerAccessCredits`. The value represents the maximum number of jobs on that particular `SimQueue` that can `START`, in other words, move from the waiting area into the service area as explained in Section 6.2. Whenever a job starts, the `serverAccessCredits` value is decremented with one, and if it reaches zero, jobs are no longer allowed to start. However, the `serverAccessCredits` value *never* affects jobs that are already in the service area.

Every `SimQueue` reports changes to *the availability of server-access credits* (i.e., not just changes to the actual value) through the `LOST_SAC` and `REGAINED SAC` notification. The former notification can be the result of starting one or more jobs *or* the invocation of `SetServerAccessCredits` with argument zero, whereas the latter notification is always the result of `SetServerAccessCredits` with argument (at least) non-zero.

In Figure 6.8, we show the modified simple model of the state of a `SimQueues` now including Server-Access Credits.

Since the number `serverAccessCredits` of server-access credits is integral, it is represented by `Java`'s `int` simple type, but the value `Integer.MAX_VALUE` is interpreted as infinity. This

Figure 6.7: `SimQueue` model with Server-Access Credits.



is in fact the default value; and as long as `ServerAccessCredits` has this value, it is not affected by starting jobs (the value is not decremented), effectively turning off the mechanism of server-access credits.

In addition to the default value being ∞, `SimQueue` implementations cannot use `ServerAccessCredits` to meet their requirements. For instance, in order to implement queueing systems with multiple servers like `FCFS_c` (see Section **??**), the use of `ServerAccessCredits` could be queue handy. However, decreasing the value upon `START` of a job is the only thing queues may do (and must adhere to).

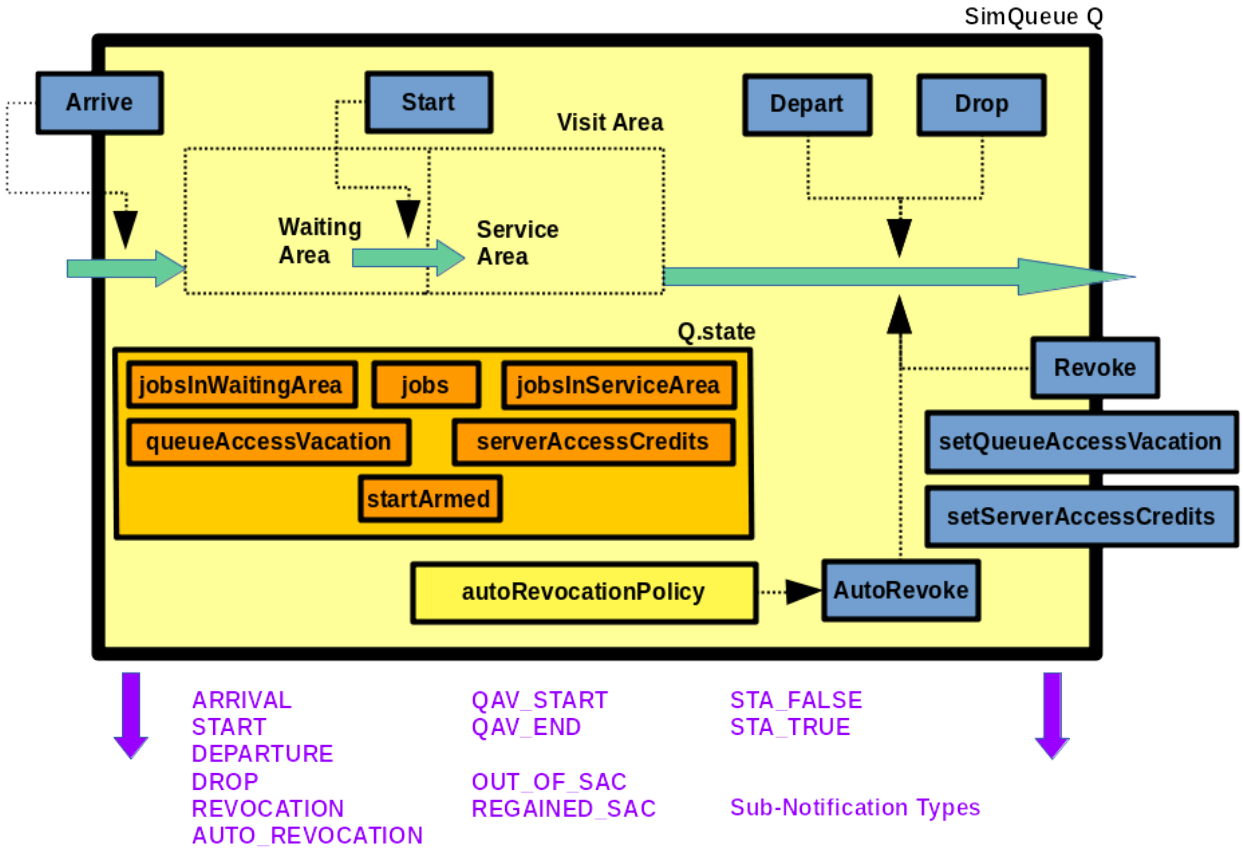
These two facts imply that if you never "touch" the `serverAccessCredits` property value through the use of the `SetServerAccessCredits` operation, you can safely forget the entire concept. On the other hand, should you have any need for it, it is always available, whatever the (concrete) queue type.

6.9 The `startArmed` State Property

Every `SimQueue` (implementation) supports the `boolean` `startArmed` state property; its changes are reported with `STA_TRUE` and `STA_FALSE`, respectively. In Figure 6.9, we show the modified simple model of the state of a `SimQueues` now including the `startArmed` state property.

The `startArmed` property is a but difficult to grasp at first sight and has little use in simulations, but it is essential in so-called *compressed tandem queues*, in particular `CTandem2` ↪ , see Section **XXX**. Nonetheless, since it is part of the `SimQueue` interface, we state its formal definition: A `SimQueue` has state property `startArmed == true` if and only if *any* (hypothetical) arriving job *would* start service immediately (i.e., enter the service area upon

Figure 6.8: SimQueue model with startArmed state property. (Full model of a SimQueue.)



arrival immediately), *if* the following conditions *would* hold:

- the absence of a queue access vacation,
- at least one server-access credit, and
- an empty waiting area.

The actual values of these three state properties is irrelevant, which, admittedly, makes the definition quite hard to understand. If a SimQueue has startArmed == true, changing its state such that it meets the three conditions above, would lead to the immediate start of a hypothetical arriving job (irrespective of the type and properties of that job).

As an example, consider an instance of the (single-server) FCFS queueing system which has no queue-access vacation, and suppose that it is out of server-access credits, has a single job in the waiting area and no jobs in the service area. In this case, startArmed == true because an arriving job would be taken into service if we apply the state transformation rules: (1) remove any queue-access vacation (check), (2) give it a single (or more) server-access credit, and (3) remove the job from the waiting area. This leaves a FCFS queueing system without queue-access-vacation, no jobs in the waiting area, and at least one server-access credits, and surely, such a queue would immediately start an arriving job. If initially, however, a job would be in service at the queue, we have startArmed == false, because the transformed state of the queueing system has a job in service, and FCFS cannot guarantee at all that an arriving job would be taken into service. If on he other side, in the latter case the queue is of type PS, we would have startArmed == true, because the presence of jobs in the service area in PS, does not inhibit it from immediately starting newly arrived jobs.

Informally, the `startArmed` state property of a queue reflects the fact *as far as the service area is concerned*, at least one more job can be added to it *immediately*.

With the introduction of the `startArmed` property, our model of a `SimQueue` is now complete.

6.10 The Requested Service Time of a Job

So far we have not been concerned with the time it takes to serve a job until completion, if a `SimQueue` supports the notion of "serving jobs". Well, in `jqueues`, the default behavior is that a queue interrogates the job for its *requested service time*. To that purpose, each `SimJob` has the `requestedServiceTimeFunction` essential property, which has type `Function<SimQueue, Double>`. Its value returns a *requested service time* for each `SimQueue` argument.

Compared to the `SimQueue` interface, the `SimJob` interface is remarkably simple. Apart from the internal maintenance of the `SimQueue` being visited, a `SimJob` only needs to provide information on the so-called *requested service time* for a queue visit, through implementation of `getServiceTime (Q)`. This method is used by a `SimQueue` to query the requested service time, and appropriately schedule a departure event for the job, but it can be called anytime. However, the returned value should not change during a visit to a `SimQueue`, and it is not manipulated by the queue being visited, in other words, it cannot be used to query the remaining service time of a job at a queue. It is safe though to change the return value in-between queue visits. However, the convention is that the method then returns the required service time at the *next* visit to the queue. For instance, many test and job-factory classes depend on this, as they often directly probe a non-visiting job for its required service time at a queue. Obviously, implementations must be prepared for invocations of this method while not visiting a queue. If `null` is passed as argument the service time at the current queue is used, or zero if the job is not currently visiting a queue.

6.11 The `DefaultSimjob` Class

The `DefaultSimJob` provides a reasonable initial implementation of `SimJob`. It features two constructors, both of which take the event list (may be `null`) and the name of the job (may be `null`) as argument. The first,

```
public DefaultSimJob (final SimEventList eventList, final String name, final Map<Q, Double>
    ↪ requestedServiceTimeMap)
```

takes as additional argument a `Map` from `Q` onto requested service times, allowing different values on a per-`SimQueue` basis. The second,

```
public DefaultSimJob (final SimEventList eventList, final String name, final double requestedServiceTime)
```

takes a `double` as third argument holding the requested service time at *any* `SimQueue`.

6.12 Summary of Properties

In Table 6.1 we list the (mandatory) properties of a `SimJob`. The table starts with a specification of the super type of `SimJob`, viz., `SimEntity`. We consistently try to always specify the super type in tables showing properties, operations, etc., merely as a reminder to the reader that such a table actually *augments* the corresponding table from the super type.

The only mandatory state property on a `SimJob` is the `queue` property which holds the `SimQueue` currently being visited by the `SimJob`. The mandatory reset value is `null`, as shown

Table 6.1: Mandatory properties of a `SimJob`.

Name	**Type**	**Default/Reset**
SimJob		
Super		
`SimEntity`		See Table 5.1.
State Properties		
`queue`	`SimQueue`	`null`.
Essential Properties		
`qosClass`	`Class`	`null`.
`qosValue`	`Object`	`null`.
`requestedServiceTimeFunction`	`Function<SimQueue, Double>`	`q->1.0;`
Cosmetic Properties		
None.		

in the third column, meaning that (1) a job cannot visit a queue "upon construction", and (2) visits to queues end upon a `Reset`.

A `SimJob`'s essential properties are the two QoS-related properties `qosClass` and `qosValue` described in Section 6.6, and the `requestedServiceTimeFunction` described in Section 6.10. Be aware that for essential (and cosmetic) properties the third column holds the property's *default* value, and that the value of such properties can *only* be set upon construction or *immediately* after a `Reset`, *unless stated otherwise*. Also, the value of an essential property typically survives a `Reset`. In the case of `SimJob`, all essential properties actually can be set safely *in between visits* as well. A `SimJob` has no specific *cosmetic* properties.

In Table 6.2 we list the (mandatory) properties of a `SimQueue`.

Table 6.2: Mandatory properties of a `SimQueue`.

Name	**Type**	**Default/Reset**
SimQueue		
Super		
`SimEntity`		See Table 5.1.
State Properties		
`jobs`	`Set<SimJob>`	Empty `Set`.
`jobsInWaitingArea`	`Set<SimJob>`	Empty `Set`
`jobsInServiceArea`	`Set<SimJob>`	Empty `Set`.
`queueAccessVacation`	`boolean`	`false`.
`serverAccessCredits`	`int`	`Integer.MAX_VALUE` ("=" ∞).
`startArmed`	`boolean`	Depends (only) on sub-type.
Essential Properties		
`autoRevocationPolicy`	`AutoRevocationPolicy`	`NONE`.
`qosClass`	`Class`	`null`.
`qosValue`	`Object`	`null`.
Cosmetic Properties		
None.		

Since `SimQueue` extends `SimEntity`, the table starts with a specification of the super type and a reference to the specification of `SimEntity`'s properties. The three visit-related state properties on a `SimQueue` are `jobs`, `jobsInWaitingArea` and `jobsInServiceArea`; note that *the former is always the disjoint union of the latter two*. In addition, any `SimJob j` present in either set on some `SimQueue q` must necessarily have `j.queue == q`.

The remaining state properties are not (directly) related to job visits; they are specific to

SimQueue. The queueAccessVacation property indicates whether or not the SimQueue is on queue-access vacation as explained in Section 6.7. The serverAccessCredits property holds the remaining number of jobs allowed to start (in absence of events changing the property). It is explained in more detail in Section 6.8. The value Integer.MAX_VALUE is interpreted in this context as infinity: If the property holds this value, it is not decremented upon the start of a job. We repeat the remark that both the queueAccessVacation and serverAccessCredits ↪ properties are always fully available to the user on *any* SimQueue implementation; they are *never* (to be) used for internal purposes. Finally, the startArmed property indicates whether or not, sloppily put, the service area would immediately admit *any* arriving job in the hypothetical case that the queue is (1) not on server-access vacation, (2) has at least one server-access credit and (3) has an empty waiting area. It is explained in detail in Section 6.9.

The mandatory cosmetic properties on a SimQueue are autoRevocationPolicy explained in Section 6.5, and the QoS-related properties qosClass and qosValue explained in Section 6.6. We repeat the remark that cosmetic properties can be changed from user code *anytime*.

6.13 Summary of Operations

In Table 6.13, we summarize the operations supported on a SimQueue, subdivided into SimEntity ↪ and SimQueue operations.

Table 6.3: The operations on a SimQueue.

SimEntity Operations		
Reset	E	double newTime
Update	E	double newTime
SimJQ Operations		
Arrive	E	double time, SimJob, SimQueue
Drop	I	double time, SimJob, SimQueue
Revoke	E	double time, SimJob, SimQueue, boolean interruptService
AutoRevoke	I	double time, SimJob, SimQueue
Start	I	double time, SimJob, SimQueue
Depart	I	double time, SimJob, SimQueue
SimQueue Operations		
SetQueueAccessVacation	E	double, SimQueue, boolean
SetServerAccessCredits	E	double, SimQueue, int

The first column in the table shows the name of the operation. Note that subtle change in naming between a *notification* (like ARRIVAL) and its corresponding *operation* (like Arrive). The second column indicates whether the operation is External (E) or Internal (I). Note that with the exception of Update, an external operation on a SimEntity is *never* invoked from within the entity itself. The third column provides the arguments to the operation, without

going into the details of the method prototypes. The argument names are only shown when needed for clarification; for most arguments, the type is self-explanatory.

6.14 Summary of Sub-Notification Types

In Table 6.14, we summarize the notification types supported on a `SimQueue`, subdivided into `SimEntity`, `SimJQ` and `SimQueue` notification types. The `SimEntity` notification types apply to any `SimEntity`, the `SimJQ` types to `SimJobs` and `SimQueues`, and the `SimQueue` types to `SimQueues` only.

Table 6.4: The sub-notification types from a `SimJob` and `SimQueue`.

`SimJQ` Sub-Notification Types (Common to `SimJob` and `SimQueue`)	
ARRIVAL	double time, SimJob, SimQueue
DROP	double time, SimJob, SimQueue
REVOCATION	double time, SimJob, SimQueue
AUTO_REVOCATION	double time, SimJob, SimQueue
START	double time, SimJob, SimQueue
DEPARTURE	double time, SimJob, SimQueue
`SimQueue` (Specific) Sub-Notification Types	
QAV_START	double time, SimQueue
QAV_END	double time, SimQueue
OUT_OF_SAC	double time, SimQueue
REGAINED_SAC	double time, SimQueue
STA_FALSE	double time, SimQueue
STA_TRUE	double time, SimQueue

The first column is the name of the sub-notification type as it appears in (for instance) the output of several `SimEntityListener` implementations. The second column provides the arguments that are supplied with the notification type; and only if needed for clarity, the argument is named. This column, however, is merely provided so you understand the meaning of the arguments in the output and in the code; it does not provide literal lists of arguments to any method. But it should, for instance, allow you to look up the `javadoc` for a specific `SimEntityListener`, and known which methods to override, and what their arguments mean. Recall that a state change is reported atomically as a `List` of sub-notifications.

6.15 Utility Methods for Scheduling

In Table 6.15, we list the most important utility methods for scheduling `SimEvents`. Note that all methods are static; you cannot instantiate the various classes.

Table 6.5: Utility Methods for Scheduling.

Class	Method (Static)
SimEntityEventScheduler	scheduleUpdate (SimEventList, SimEntity, double)
SimJQEventScheduler	scheduleJobArrival (SimJob, SimQueue, double)
SimJQEventScheduler	scheduleJobRevocation (SimJob, SimQueue, double, boolean)
SimQueueEventScheduler	scheduleQueueAccessVacation (SimQueue, double, boolean)
SimQueueEventScheduler	scheduleServiceAccessCredits (SimQueue, double, int)

6.16 Atomicity and Invariants

If you have read this tour linearly, and have reached this point, you have mastered most of the essential features of `jqueues` and `jsimulation`. In the current and next section, we cover two aspects that you may want to skip at first reading, viz., *composite* queues in the next section, and *atomicity* and *queue invariants* in this section. They are, however, signature features of `jqueues`.

As mentioned many times earlier, every `SimEntity` supports a well-defined set of operations. These operations, at least the external ones, are invoked from the event list, or otherwise from within the `SimEntity` itself while processing another operation. The state of a `SimEntity` ↪ can only change as a result of the invocation of an operation on it. Immediately after an operation invocation has been received, the `SimEntity` must fire an `UPDATE` notification to its registered listeners, exposing the entity's state immediately prior to receiving the invocation. After completion of the operation, the `SimEntity` must fire a `STATE CHANGED` notification, exposing its new state. The notification may sometimes be left out if the state did not actually change, but, many operation invocations are *always* reported to listeners, even if they do not result in state changes (for instance, invocations of `Arrive`, `Drop`, `Start` and `DEPART` on a `SimQueue`).

But this you already knew, right? So, there has to be more... Let's do this, one step at a time.

First, forget about *all* other notifications than `RESET`, `UPDATE` and `STATE CHANGED`. All other forms or notifications are essentially *courtesy* notifications derived from (mostly) `STATE` ↪ `CHANGED` notifications. You may have already discovered that such courtesy notifications are always fired *after* the `STATE CHANGED` notifications, if at all. It is essential to realize that a full description of the state changes is always available in the `STATE CHANGED` notification. It is often harder to process than the courtesy notifications, sure, but all courtesy notifications are in essence derived from it.

Second, `STATE CHANGED` notifications consist of a *sequence of sub-notifications*. Because, often, the invocation of an event leads to a more complicated state change than can be described in a single notification: Jobs might start upon arrival, or be dropped due to queue-access vacations, or even depart immediately upon arrival. This explains why a `STATE CHANGED` ↪ notification comes with a `Set` of `SimEntityEvents` (in proper sequence), each describing a single step in the state transition.

Third, realize that invocations of an operation come in two flavors: An initial invocation from the event list, and, if applicable, subsequent invocations while processing that initial invocation. (These subsequent invocations are almost always for *internal* operations, but this is by no means a requirement of even the general case.) We refer to the initial revocation as the *top-level operation (invocation)*, and the others as *chained operation invocations*, or shortly (and sloppily) *chained operations*.

We have now set the stage for the big one: *Top-level operation invocations are processed*

atomically. Meaning? Well, in simple terms, only the top-level operation is allowed to notify listeners, and only after the new state has been achieved completely. So, upon completion of a chain operation, its relevant notifications are deferred until the top-level notification has finished completely.

Why should we care? Well, consider the following example: We schedule at $t = 0$, a single arrival of a single job requiring 1.0 service time at an otherwise empty `FCFS` queue. There are no queue-access vacations, and the queue has unlimited server-access credits (both are default settings). When the event list processes the event, we know what happens: The job arrives, enters the waiting area, and, because the server is idle, the job starts immediately. Nothing new... But: The scheduled `Arrive` operation is top-level, the resulting `Start` operation is chained, and therefore not allowed to issue notifications by itself. This means that upon return from the `Arrive` operation, we get a combined (using an improvised notation) `STATE CHANGED` ↪ `[ARRIVAL, START]` notification, instead of two separate ones, `STATE CHANGED[ARRIVAL]` followed by `STATE CHANGED[START]`.

So what is the big deal? Well, in order to appreciate atomicity of top-level operation invocations, we need the concept of a *queue invariant*, being a statement about a queue (`SimQueue`) that should *always* hold. In this particular case for `FCFS`, we do not expect jobs residing in the waiting area if the service area is empty (and server-access credits are available). Hence, a properly formulated queue invariant for `FCFS` (and many other queueing systems) is: *If the service area is empty and the queue has non-zero server-access credits, then its waiting area is empty.* In other words: "There cannot be waiting jobs if the server is idle".

So, continue please? Well, in `jqueues`, *queue invariants are always* `true` *when notifications are fired.* This is an essential, and in many ways distinguising feature of `jqueues`. It means that `SimQueue` implementations can document their invariants, and listeners are sure that these invariants are always met. This is hard if not impossible (in general) without requiring atomicity on the invocation of operations.

In order to appreciate this, let's consider an alternative approach to implementing `SimQueue` (well, in fact, let's consider a different *interface*). The straightforward strategy we actually used in earlier versions of `jqueues` is to always *schedule* the `Start` operation on the event list on a `FCFS` queue. The implementation is really a whole lot simpler: Upon the arrival of a job, the job is added to the tail of the waiting area and if it is the only one in the waiting area, a `Start` event is scheduled on the event list, obviously with the same event time as the arrival time. The scheduled `Start` operation, once invoked, takes the job out of the waiting area, adds it to the service area, and schedules the invocation of its `Depart` operation. The `Depart` operation then removes the job from the service area, as well as taking other actions to end the visit. And, lo and behold, the approach works just fine. So simple... But: The `Arrive` and `Start` operation must obviously report their inflicted state changes to the registered listeners. This means that `Arrive` notifies its listeners with an `ARRIVAL` notification with the newly arrived job's shining presence in the waiting area. Sure, it will be gone a jiffie (not even that much) later, but the listeners are exposed to a state that is a violation of an all but reasonable queue invariant. It exposes the job in the waiting area while the service area is empty...

So, you create a much more complicated interface and implementation of a `SimQueue` (even of a `SimEntity`) just to achieve that a queue does not expose itself to listeners in a state inconsistent with the queue invariant? Are they that important? Well, yes! They are really that important...

By the way, the `FCFS` queue has an additional queue invariant: its service area does not contain jobs with zero required service time. So if the job in the example would indeed require no service, the notification in response to the invocation of the `Arrive` operation would indeed be `STATE CHANGED[ARRIVAL, START, DEPARTURE]`.

The realization that top-level events (again, sloppy) should be processed atomically was

of critical importance for the implementation of *composite queues*; queues that consists of the combination of other queues, the so-called sub-queues. We will introduce them in the next section.

Chapter 7

Fundamental Queues

In this chapter, we introduce *all* `SimQueue` implementations for *fundamental* queueing systems in `jqueues` Release 5. A queueing system is named *fundamental* if it is not constructed by composition of other queueing system instances as in for instance tandem, feedback and Jackson queueing systems. (This is a `jqueues`-specific definition.) Such *composite* queueing systems are introduced and described in detail in the next chapter. In the present chapter, we describe the queueing systems to the extent that they can be put to use in a simulation. It does not aim to be complete in the individual descriptions, however.

7.1 Serverless Queues

In so-called *serverless* queues, jobs never enter the service area, in other words, *they never start*. As a result, the service area of a serverless queue is always empty, server-access credits are ignored, and the `startArmed` property is always `false`. In Table 7.1, we list all serverless queues implemented in `jqueues`, accompanied with a brief description.

Table 7.1: The serverless queueing systems in `jqueues` Release 5 at a glance.

Serverless Queues	
`DROP`	*Drops jobs immediately upon arrival.*
`SINK`	*Lets jobs wait indefinitely.*
`ZERO`	*Lets jobs depart immediately upon arrival.*
`DELAY`	*Lets jobs depart after a fixed wait time.*
`GATE`	*Lets jobs depart immediately while the queue has "gate-passage credits".*
`ALIMIT`	*Drops (admitted) jobs if needed to not exceed a given arrival-rate limit.*
`DLIMIT`	*Lets jobs wait if needed to not exceed a given departure-rate limit.*
`LeakyBucket`	*Lets jobs wait if needed to not exceed a given departure-rate limit; the waiting area has finite size, though.*
`WUR`	*Lets jobs wait until the next (admitted) arrival.*

It is important to realize that despite the virtual absence of the service area, a serverless

queue still reports running out of and regaining server-access credits, conform Section 6.8. In addition, all queues support queue-access vacations, as outlined in Section 6.7, and job revocations described in Section 6.4.

The `DROP`, `SINK`, `ZERO` and `WUR` queueing systems hopefully speak for themselves. These do not have additional properties, operations, or sub-notification types in addition to the bare `SimQueue` interface.

The `DELAY` queue lets jobs depart (from the waiting area) a fixed time after their arrival, captured in the queue's `waitTime` essential property with type `double`. Setting the property value to zero turns `DELAY` into a `ZERO`; setting it to infinity (`Double.POSITIVE_INFINITY`) turns it into a `SINK`.

The `GATE` queueing system is a bit more involved. It features a `gatePassgeCredits` state property of type `int` holding the remaining number of jobs allowed to depart in order of arrival. On each departure, the property value is decremented, and if it reaches zero, jobs are no longer allowed to depart; they must reside in the waiting area instead. However, the value `Integer.MAX_VALUE`, the reset value, is treated as infinity; prohibiting the decrements upon departures. If the property value reaches zero, a `GATE_CLOSED` sub-notification is issued; if credits are regained a `GATE_OPEN`. The property value can be set through the external `SetGatePassageCredits` operation. Note the high degree of similarity with a `SimQueue`'s server-access credits with `GATE`'s gate-passage credits. Informally, the latter replaces the former in view of the lack of starting jobs.

The `ALIMIT`, `DLIMIT` and `LeakyBucket` queues impose limits on arrival (for `ALIMIT`) or departure (for `DLIMIT` and `LeakyBucket`) rates. Each is equipped with a `rateLimited` state property of `boolean` type indicating that the queue is currently in *rate limitation*, as well as a `rateLimit` essential property of type `double`. In the current release, changes to the `rateLimited` state property value are *not* reported through dedicated sub-notification types, but since this is a violation of the `SimQueue` contract (unless we think of the property as "internal"), this will be fixed in a future release.

The `ALIMIT` queue drops arriving jobs while it is in rate limitation, whereas both `DLIMIT` and `LeakyBucket` put arriving jobs into the waiting area until the rate limitation has expired. (This is in fact accomplished by an anonymous internal operation.) The only difference between the latter two is that `LeakyBucket` has a waiting area *of finite size* as mandated by its `bufferSize` essential property. It is of type `int`, and, as usual, the value `Integer.MAX_VALUE` is treated as infinity. The `LeakyBucket` queue drop jobs upon arrival if it is in rate limitation *and* there are already `bufferSize` other jobs present in the waiting area.

For `ALIMIT`, rate limitation starts upon an admitted arrival[1], and always ends 1/`rateLimit` after that. As a special case, setting `rateLimit` to zero effectively turns `ALIMIT` into a `DROP` ↪ ; setting it to infinity turns `ALIMIT` into a `ZERO`. Note that whatever the property value, the `ALIMIT` queue *never* has jobs in its waiting area (or its visit area, for that matter). For `DLIMIT` and `LeakyBucket`, rate limitation *always* starts or continues at a departure; it lasts for 1/`rateLimit`, *if* at the projected end time the waiting area is empty. Otherwise, the job longest in the waiting area departs, restarting rate limitation for another 1/`rateLimit`.

Finally, a noteworthy disambiguation of `LeakyBucket` with zero `bufferSize` is that, shortly put, absence of rate limitation takes precedence over the absent buffer space. This means that *if* `LeakyBucket` is *not* in rate limitation, arriving jobs will depart immediately (starting a new rate-limitation period).

[1] In this context, an arriving job that is not dropped due to queue-access vacation *or* due to rate limitation

7.2 Non-Preemptive Queues

In `jqueues`, so-called *non-preemptive* queues serve jobs in their service area until they depart. Jobs *never* depart from the waiting area. Non-preemptive queues in `jqueues` differ in (1) the size of the waiting area (`bufferSize`), (2) the number of (always equal) servers in the service area (`numberOfServers`), (3) the order in which jobs are selected for service from the waiting area and/or (4) the selection policy for dropping jobs in case the number of jobs waiting is about to exceed the buffer size. In Table 7.2, we list all non-preemptive queues implemented in `jqueues`, accompanied with a brief description. The queues are presented in several groups, each with important common structure and behavior.

Table 7.2: The non-preemptive queueing systems in `jqueues` Release 5 at a glance.

Non-Preemptive Queues	
	Serves jobs in order of arrival until completion as defined by the jobs' requested service times...
`FCFS`	*- with a single server and infinite buffer size.*
`FCFS_B`	*- with a single server and finite buffer size.*
`FCFS_c`	*- with multiple servers and infinite buffer size.*
`FCFS_B_c`	*- with multiple servers and finite buffer size.*
`NoBuffer_c`	*- with multiple servers and no buffer at all.*
`IS`	*- with infinite servers and infinite buffer size.*
	Serves jobs in reverse order of arrival until completion as defined by the jobs' requested service times...
`LCFS`	*- with a single server and infinite buffer size.*
`LCFS_B`	*- with a single server and finite buffer size.*
	Serves jobs in order of arrival...
`IC`	*- in zero service time with infinite buffer size.*
`IS_CST`	*- in constant service time with infinite buffer size.*
	Serves jobs until completion with a single server as defined by the jobs' requested service times...
`SJF`	*- in order of increasing requested service time...*
`LJF`	*- in order of decreasing requested service time...*
`RANDOM`	*- in random order...*
	with infinite buffer size.
`SUR`	*Serves jobs until the next (admitted) arrival with a single server and infinite buffer size.*

Before going into details on the respective groups of non-preemptive queueing systems, it is important to realize that each of them is subject to the mechanism of server-access credits explained in Section 6.8. Also, each has at least two essential properties: (1) `bufferSize` limits the number of jobs in the waiting area and (2) `numberOfServers` limits the number of jobs

in the service area. Both are of type `int`, and as usual, `Integer.MAX_VALUE` is interpreted as infinity.

The center of gravity, i.e., the most general form, of the first group is the `FCFS_B_c` queueing system, featuring user-settable `bufferSize` (`B`) and `numberOfServers` (`c`) essential properties. All other members in this group are specializations of `FCFS_B_c` fixing one or both properties. In `FCFS_B_c`, in absence of queue-access vacations, an arriving job is taking into service *immediately* (1) the queue has non-zero server-access credits, *and* (2) the queue has strictly fewer jobs in the service area than the value of the `numberOfServers` property (respecting the convention that `Integer_MAX_VALUE` represents infinity). Otherwise, the arriving job is entered into the waiting area of the queue (at the tail), provided that the current number of jobs in the waiting area is strictly smaller that the `bufferSize` property. If this is not the case, the arriving job is dropped.

In `FCFS_B_c`, the other epochs at which jobs are eligible to start are (1) the granting of server-access credits, (2) the scheduled departure of a job (from the service area) or (3) the revocation of a job from the service area. In any of these cases, if there are server-access credits, servers available (respecting the case of an infinite number of servers) and jobs in the waiting area, the job in the waiting area with the earliest arrival time is started, i.e., admitted to the service area. Once admitted to the service area, a job is served until completion as prescribed by its requested service time, see Section 6.10, unless it is somehow revoked.

The second group of non-preemptive queues listed in Table 7.2, centered around `LCFS_B`, differs from the first group in only two ways:

- In case of depletion of buffer space, the job that arrived *earliest* is dropped (instead of the arriving jobs).

- The service area selects the job that arrived latest as the eligible job to start.

The remaining groups in Table 7.2 speak for themselves; since each queue has infinite buffer space, we do not have to worry about job drops. The only things different from the previous groups are the selection of jobs for admission to the service area or the time required to serve a job once admitted to the service area.

7.3 Preemptive Queues

Unlike non-preemptive queueing systems described in the previous section, preemptive queueing systems do not always provide service to jobs in the service area, and may in fact interrupt (*preempt*) execution of a job in favor of another job (in the service area). In Table 7.3, we list all preemptive queues implemented in `jqueues`, accompanied with a brief description.

At the moment a job is preempted at a server, one of several things can happen, as determined by the `preemptionStrategy` essential property. In `jqueues` Release 5, the preemption strategy is of type `enum` named `PreemptionStrategy`, and it can take the following values:

- `DROP`: The preempted job is dropped.

- `RESUME`: The preempted job is put on hold, after calculating the remaining service time. Future service resumption continues at the point where the previous service was interrupted. (This is the default.)

- `RESTART`: The preempted job is put on hold. Future service resumption requires the job to be served from scratch.

Table 7.3: The preemptive queueing systems in `jqueues` Release 5 at a glance.

Preemptive Queues	
	Serves jobs until completion, as defined by the jobs' requested service times, or until preemption with a single server and infinite buffer size...
`P_LCFS`	*- in reverse arrival order.*
`SRTF`	*- in increasing order of remaining service time.*

- `REDRAW`: The preempted job is put on hold. Future service resumption requires the job to be served from scratch with a new required service time. (This is not supported in `jqueues` Release 5.)
- `DEPART`: The preempted job departs, even though it may not have finished its service requirements.
- `CUSTOM`: A different strategy that those mentioned above is applied to preempted jobs. The strategy needs to be specified in the concrete `SimQueue` implementation.

Not all preemptive-queue implementations support all preemption strategies. For instance, the `REDRAW` strategy is not supported by any implementation, because it conflicts with the contract that the requested service time of a `SimJob` has to remain constant during a (single) `SimQueue` visit.

The preemption strategy is a parameter that can be passed upon construction to all implemented preemptive queues in `jqueues`. Its default value is `RESUME`. If you pass an unsupported preemption strategy, the queue will throw an `UnsupportedOperationException`.

For consistency with non-preemptive queueing systems, both `P_LCFS` and `SRTF` support the `bufferSize` and `numberOfServers` essential properties, but their value are fixed at `Integer.` ↪ `MAX_VALUE`, representing infinity, and unity, respectively.

7.4 Processor-Sharing Queues

In the preceding two sections on non-preemptive and preemptive queueing systems, respectively, a common characteristic was that the service area consists of a countable number of servers, each serving at most one job, but not necessarily *all job* in the service area (in particular, this was found not to be true in general for preemptive queues). In so-called *processor-sharing* queueing systems, most of this model still holds, with the exception that the a single server is capable of providing service to *multiple* jobs simultaneously, distributing its total service capacity among them. They are of extreme theoretical importance in queueing theory, because they often can be regarded as the limiting case of "time sharing": Serving a single job for small amounts of time, then saving its state and resuming service to another job from its saved state. In Table 7.4, we list all processor-sharing queues implemented in `jqueues`, accompanied with a brief description.

The three processor-sharing queues have in common that they consist of a single server in the service area, and that in presence of a server-access credit, and arriving job is admitted

Table 7.4: The processor-sharing queueing systems in `jqueues` Release 5 at a glance.

Processor-Sharing Queues	
	Serves simultaneously with a single server and infinite buffer size, distributing its capacity among jobs in the service area...
`PS`	*- at equal rates.*
`CUPS`	*- with the lowest obtained service time at equal rates.*
`SocPS`	*- at rates linearly proportional to their remaining service times.*

immediately to the service area. The three systems only differ in the way the server distributes its capacity among jobs in the service area.

The `PS` implements the (Egalitarian) Processor Sharing queueing discipline with a single server. It is probably well-known to many readers. In `PS`, the server simply distributes its capacity equally among *all* jobs in the service area.

In `CUPS` (Catch-Up Processor Sharing), the server distributes its capacity equally among all jobs in the service area with the lowest *obtained* service time. Whenever a job starts, it enters the service area with zero obtained service time, and assuming there are no new arrivals after that, it will therefore be served exclusively. The jobs that were served a priori, will have to wait in the service area until the new job has reached sufficient service to "catch up" and join that set. From that point on, the server will distribute its capacity among the "a priori set of jobs" augmented with the newly arrived job. The resulting set of jobs will be served at equal rates until they catch up with the next set of jobs, etc. In case a job has received sufficient service as mandated by its requested service time, it will leave the set of "jobs in service", and if that set turns empty because of that, `CUPS` will switch to the next set of jobs with minimum obtained service time, if any. If not, the server will turn idle. Intuitively, `CUPS` gives extreme preferential treatment to jobs with small requested service time. To our knowledge, the `CUPS` queueing system has not been described in queueing-system literature before[2].

In `SocPS` (Social Processor Sharing), the single server distributes its capacity among jobs in the service area at rate linearly proportional to their *remaining service time*. This implies that, ignoring for instance revocations, *all jobs in the service area will depart at the same time.* This includes jobs with zero requested service time; they will simply not be served if other jobs with non-zero remaining service times are present in the service area. As `CUPS`, `SocPS` is a queueing discipline we have not found earlier in literature.

For consistency with non-preemptive and preemptive queueing systems described earlier, the processor-sharing implementations support the `bufferSize` and `numberOfServers` essential properties, but their value are fixed at `Integer.MAX_VALUE`, representing infinity, and unity, respectively.

7.5 QoS Queues

We end this chapter with a separate presentation of QoS queueing systems, even though in `jqueues` each of the QoS queueing systems could be categorized in one of the earlier described classes (non-preemptive, preemptive and processor-sharing queueing systems) as well. Recall

[2]But we have not fully checked this, and would be highly interested in relevant references.

from Section 6.6 that in such systems, visiting jobs are treated differently corresponding to their membership of some "group" of jobs[3]. To that purpose, jobs and queues need non-`null` and compatible values for their `qosClass` property; in addition, each job needs a value for its `qosValue` property, indicating the "group" of jobs to which it belongs. In Table 7.5, we list all QoS queues implemented in `jqueues`, accompanied with a brief description.

Table 7.5: The QoS queueing systems in `jqueues` Release 5 at a glance.

QoS Queues	
`HOL`	*Serves jobs until completion in order of increasing* `QoS` *values with a single server and infinite buffer size. Serves jobs identical* `QoS` *values in arrival order. The server can serve at most one job at a time; it does not preempt jobs.*
`PQ`	*Serves jobs until completion or preemption in order of increasing* `QoS` *values with a single server and infinite buffer size. Preempts a job in service upon the start of another job with strictly smaller* `QoS` *value. Serves jobs with identical* `QoS` *values in arrival order. The server can serve at most one job at a time.*
`HOL_PS`	*Serves simultaneously (equally distributing its capacity) all earliest-arrived jobs with distinct* `QoS` *values with a single server and infinite buffer size. Serves jobs with identical* `QoS` *values in arrival order.*

From their brief description, one deduces that `HOL` is a non-preemptive queueing system, and that `PQ` and `HOL_PS` are preemptive and processor-sharing ones, respectively. For `HOL` and `PQ`, the `qosClass` used must support (at least) partial ordering of the values used. Therefore, the `qosClass` must be `Comparable` in the `Java` sense. For `HOL_PS`, this is not required, any `class` or `interface` will do; the `qosValue` property values are compared using the `Object.equals` method, following the convention in `Java`'s `Set` interface in the `Collections` framework.

Since `PQ` is a preemptive queueing system, it supports the preemption strategy mechanism introduced in Section 7.3.

[3]We avoid the more common "job class" terminology since it is highly confusing in the `jqueues` context.

Chapter 8

Composite Queues

In the preceding sections, we have already seen some example queueing systems like `FCFS_B`, `FCFS_c` and `PS`, all of which implement the `SimQueue` interface. There are many more, all of which are described in detail in later chapters of this book. One particular "class" of `SimQueue` implementations, however, are the so-called *composite* queues, and they deserve mentioning in this guided tour.

A composite queue is a "queue of queues"; its behavior is determined by a set of other (possibly, again composite) queues, the *sub-queues*, passed at construction, and rules for routing jobs through the network of sub-queues. These rules depend on the type of composite queue.

Sounds complicated? Well, it's really not that bad. Let's just dive into it, by considering a classic example of a composite queue: the *tandem queue*. In a tandem queue, two or more queues are concatenated in the sense that an arriving job must first visit the first sub-queue, and, upon completion at the first, visit the second queue, and so on. In Listing 8.1, we show how to create a `Tandem` with two sub-queues, a `FCFS` followed by a `DLIMIT` queue.

Listing 8.1: A tandem of `FCFS` and `DLIMIT`.

```
final SimEventList el = new DefaultSimEventList ();
final SimQueue fcfs = new FCFS (el);
// Uncomment to print events at fcfs.
// fcfs.registerSimEntityListener (new StdOutSimEntityListener ());
final double rateLimit = 0.25;
final SimQueue dlimit = new DLIMIT (el, rateLimit);
// Uncomment to print events at dlimit.
// dlimit.registerSimEntityListener (new StdOutSimEntityListener ());
final Set<SimQueue> subQueues = new LinkedHashSet<> ();
subQueues.add (fcfs);
subQueues.add (dlimit);
final SimQueue tandem = new Tandem (el, subQueues, null);
tandem.registerSimEntityListener (new StdOutSimEntityListener ());
for (int j = 1; j <= 2; j++)
{
  final double jobServiceTime = 1.5;
  final double jobArrivalTime = (double) j;
  final String jobName = Integer.toString (j);
  final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
  SimJQEventScheduler.scheduleJobArrival (job, tandem, jobArrivalTime);
}
el.run ();
```

In the example, we use a `LinkedHashSet`, instead of for instance a `HashSet`, to pass to the constructor of `Tandem`. The is essential because iteration over the sub-queues in the set must return the sets in proper order; we want `FCFS` first, then `DLIMIT`. It is recommended to always use `LinkedHashSet` for passing the set of sub-queues.

The `Tandem` queue has a third argument to its constructor, the so-called *delegate-job factory*. Because `Tandem` is in itself also a `SimQueue`, it must follow the rules of that interface, one of them being that a `SimJob` *can only visit a single* ***`SimQueue`*** *at a time*. Because the arriving job already visit the tandem queue, it cannot also visit any of the sub-queues at the same time; the tandem queue must therefore create a new `SimJob` for every job visit for routing through

the sub-queues. We call the jobs visiting the composite queue "real jobs", and the jobs created by the composite queue and visiting the sub-queues, "delegate jobs". The third argument to the constructor of `Tandem` (and the last argument to the constructor of most other concrete composite queues) allows passing a factory for delegate jobs. Passing `null` implies that the composite queue resorts to a default `DelegateSimJobFactory`, which should do just fine in most practical cases.

Listing 8.2: The output of Listing 8.1.

```
StdOutSimEntityListener t=1.0, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=1.0, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => ARRIVAL [Arr[1]@Tandem[FCFS,DLIMIT[0.25]]]
  => START [Start[1]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=2.0, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=2.0, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => ARRIVAL [Arr[2]@Tandem[FCFS,DLIMIT[0.25]]]
  => START [Start[2]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=2.5, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=2.5, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => DEPARTURE [Dep[1]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=4.0, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=6.5, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=6.5, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => DEPARTURE [Dep[2]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=10.5, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
```

Before looking at the output of the program in Listing 8.2, we need to clarify the operation of `DLIMIT`, the departure-rate limiter. It is described in detail in Section **??**. `DLIMIT` is serverless (it never starts jobs), and lets jobs depart in order of arrival immediately, yet it guarantees a minimum time (the reciproke of the rate limit argument) between its successive departures, letting jobs wait for departure only when needed. So, indeed, `DLIMIT` does what it promises to do: ensuring that the rate of departures does not exceed the rate-limit argument given.

In the example code in Listing 8.1, we schedule two job arrivals, one at $t = 1$ and the other at $t = 2$, each job requiring 1.5 service time. Since our `FCFS` queue preceeds the `DLIMIT` queue, the arriving jobs (well, in fact, their delegate jobs) immediately arrive at `fcfs`. It is trivial to check that job "1" departs from `fcfs` at $t = 2.5$, and job "2" at $t = 4.0$. Hence, by virtue of `Tandem`, at $t = 2.5$, job "1" arrives at `dlimit` and since that queue has not seen any jobs yet, it lets job "1" depart immediately. As a result, job "1" (the "real" job this time) departs from `tandem` at $t = 2.5$ The `DLIMIT` queue, however, "blocks" departures for 4 seconds starting at $t = 2.5$ in order to meet its rate-limit requirement. Therefore, job 2, arriving at `dlimit` at $t = 4.0$, must wait until $t = 2.5 + 4.0 = 6.0$ before it can depart from `dlimit`, and, as a result, from `tandem` as well.

Note that `tandem` hides the arrival of job "2" at `dlimit`. This is typical for composite queues; they hide all events occuring at their sub-queues if they do not change the state of the composite queue itself. If you want to follow what happens at both sub-queues, uncomment the marked lines in Listing 8.1, thereby registering suitable listeners at them.

Our next (and final) example of a composite queue concerns a so-called *feedback queue*, in which departing jobs are "fed back" as arrivals depending on some *feedback condition*. Unlike `Tandem`, feedback queues only allow a single sub-queue. In `jqueues` Release 5, two types of tandem queues have been implemented, viz., `FB_v` and `FB_p`. In the former, jobs have to visit the sub-queue a fixed number of times before they leave the composite queue. In the latter, jobs, upon departure from the sub-queue, are fed back with a given, fixed, probability. The example in Listing 8.3 with output in Listing 8.4 demonstrates the use of feedback queues. In the example, we create both a `FB_v` queue and a `FB_p` queue, and schedule a single arrival at each of them. The final argument to the constructors of both `FB_v` and `FB_p` is, again, the optional delegate-job factory. The fourth argument to the constructor of `FB_p` is an optional (`Java`) `Random` object in case you want to control the random-number generation (e.g., if you want to set the *seed* of the generator). The example also shows a nice way to directly schedule

actions (other than operations on queues and jobs) on a `SimEventList`. Refer to Chapter ?? for further reading if you are curious about this and other constructs on an event list.

Listing 8.3: Two feedback queues each with a `FCFS` sub-queue.

```
// Create the event list.
final SimEventList el = new DefaultSimEventList ();
// FB_v example; scheduled at t=0.
final SimQueue fcfs1 = new FCFS (el);
final int numVisits = 7;
final SimQueue fb_v = new FB_v (el, fcfs1, numVisits, null);
fb_v.registerSimEntityListener (new StdOutSimEntityListener ());
final SimJob job1 = new DefaultSimJob (null, "1", 1.0);
SimJQEventScheduler.scheduleJobArrival (job1, fb_v, 0.0);
// Create some vertical space in the output at t=9.
el.schedule (9.0, (SimEventAction) (final SimEvent event) ->
{
  System.out.println ();
});
// FB_v example; scheduled at t=10.
final SimQueue fcfs2 = new FCFS (el);
final double pFeedback = 0.8;
final SimQueue fb_p = new FB_p (el, fcfs2, pFeedback, null, null);
fb_p.registerSimEntityListener (new StdOutSimEntityListener ());
final SimJob job2 = new DefaultSimJob (null, "2", 1.0);
SimJQEventScheduler.scheduleJobArrival (job2, fb_p, 10.0);
// Run the event list.
el.run ();
```

Note that you cannot "reuse" sub-queues in multiple composite queues; in the example we cannot create a single `FCFS` queue and pass it to both `fb_v` and `fb_p`. In effect, by passing a sub-queue to a composite queue, you pass ownership over the sub-queues to the composite queue. Also, obviously, you should not schedule events at sub-queues! Most composite-queue implementations are equipped with insane sanity checks for this, and will happily throw an exception at you if they detect unexpected events at one of their sub-queues.

Listing 8.4: The output of Listing 8.3.

```
StdOutSimEntityListener t=0.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=0.0, entity=FB_7[FCFS]: STATE CHANGED:
  => ARRIVAL [Arr[1]@FB_7[FCFS]]
  => START [Start[1]@FB_7[FCFS]]
StdOutSimEntityListener t=1.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=2.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=3.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=4.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=5.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=6.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=7.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=7.0, entity=FB_7[FCFS]: STATE CHANGED:
  => DEPARTURE [Dep[1]@FB_7[FCFS]]

StdOutSimEntityListener t=10.0, entity=FB_80.0%[FCFS]: UPDATE.
StdOutSimEntityListener t=10.0, entity=FB_80.0%[FCFS]: STATE CHANGED:
  => ARRIVAL [Arr[2]@FB_80.0%[FCFS]]
  => START [Start[2]@FB_80.0%[FCFS]]
StdOutSimEntityListener t=11.0, entity=FB_80.0%[FCFS]: UPDATE.
StdOutSimEntityListener t=12.0, entity=FB_80.0%[FCFS]: UPDATE.
StdOutSimEntityListener t=12.0, entity=FB_80.0%[FCFS]: STATE CHANGED:
  => DEPARTURE [Dep[2]@FB_80.0%[FCFS]]
```

This concludes our examples on composite queues. There is really a lot more to discuss, but for this initial tour, we leave it at this. We hope you are curious to the answers to question like

- When does a real job start?
- How do server-access credits work on a composite queue?

8.1 Introduction

Composite queues consist of zero or more other queues named *subqueues* or *embedded queues*, and a visit of a job to the composite queue is equivalent to a sequence of visits to the subqueues,

the order of which is determined by the composite queue. The most prominent example of a composite queue is called a *tandem queue*, consisting of a finite sequence of (distinct) queues that a job must visit in order before leaving the composite queue. If we number the subqueues $1, 2, \ldots, K$, a job visiting the tandem queue, must first complete a visit to queue 1, and upon departure from queue 1, it immediately arrives at queue 2, and so forth, until it departs from queue K, at which time it leaves the tandem queue. Using the previous chapters, it is actually rather easy to construct a tandem queue, e.g., a tandem queue consisting of two `P_LCFS` queues, as we have shown in Listing 8.5 below. The output of the program is shown in Listing 8.6.

Listing 8.5: A tandem queue consisting of two P_LCFS queues (using SimJob).

```
private static final class TandemJob
extends DefaultSelfListeningSimJob
{

  final List<SimQueue> queues;

  public TandemJob (final SimEventList eventList,
    final String name,
    final double requestedServiceTime,
    final List<SimQueue> queues)
  {
    super (eventList, name, requestedServiceTime);
    this.queues = queues;
    this.registerSimEntityListener (new StdOutSimEntityListener ());
  }

  @Override
  public void notifyDeparture (final double time,
    final DefaultSelfListeningSimJob job,
    final SimQueue queue)
  {
    if (this.queues.indexOf (queue) < this.queues.size () - 1)
      getEventList ().schedule (time, (SimEventAction) (SimEvent event) ->
    {
      queues.get (queues.indexOf (queue) + 1).arrive (time, job);
    });
  }

}

public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  final P_LCFS lcfs_1 = new P_LCFS (el, null);
  lcfs_1.setName ("Q1");
  final P_LCFS lcfs_2 = new P_LCFS (el, null);
  lcfs_2.setName ("Q2");
  final List<SimQueue> queueSequence = new ArrayList<> ();
  queueSequence.add (lcfs_1);
  queueSequence.add (lcfs_2);
  for (int j = 1; j <= 5; j++)
    lcfs_1.scheduleJobArrival (j, new TandemJob (el, "J" + j, 10 * j, queueSequence));
  el.run ();
}
```

Listing 8.6: Output of the program in Listing 8.5.

```
StdOutSimEntityListener t=1.0, queue=Q1: ARRIVAL of job J1.
StdOutSimEntityListener t=1.0, queue=Q1: START of job J1.
StdOutSimEntityListener t=2.0, queue=Q1: ARRIVAL of job J2.
StdOutSimEntityListener t=2.0, queue=Q1: START of job J2.
StdOutSimEntityListener t=3.0, queue=Q1: ARRIVAL of job J3.
StdOutSimEntityListener t=3.0, queue=Q1: START of job J3.
StdOutSimEntityListener t=4.0, queue=Q1: ARRIVAL of job J4.
StdOutSimEntityListener t=4.0, queue=Q1: START of job J4.
StdOutSimEntityListener t=5.0, queue=Q1: ARRIVAL of job J5.
StdOutSimEntityListener t=5.0, queue=Q1: START of job J5.
StdOutSimEntityListener t=55.0, queue=Q1: DEPARTURE of job J5.
StdOutSimEntityListener t=55.0, queue=Q2: ARRIVAL of job J5.
StdOutSimEntityListener t=55.0, queue=Q2: START of job J5.
StdOutSimEntityListener t=94.0, queue=Q1: DEPARTURE of job J4.
StdOutSimEntityListener t=94.0, queue=Q2: ARRIVAL of job J4.
StdOutSimEntityListener t=94.0, queue=Q2: START of job J4.
StdOutSimEntityListener t=123.0, queue=Q1: DEPARTURE of job J3.
StdOutSimEntityListener t=123.0, queue=Q2: ARRIVAL of job J3.
StdOutSimEntityListener t=123.0, queue=Q2: START of job J3.
StdOutSimEntityListener t=142.0, queue=Q1: DEPARTURE of job J2.
StdOutSimEntityListener t=142.0, queue=Q2: ARRIVAL of job J2.
StdOutSimEntityListener t=142.0, queue=Q2: START of job J2.
StdOutSimEntityListener t=151.0, queue=Q1: DEPARTURE of job J1.
StdOutSimEntityListener t=151.0, queue=Q2: ARRIVAL of job J1.
StdOutSimEntityListener t=151.0, queue=Q2: START of job J1.
StdOutSimEntityListener t=161.0, queue=Q2: DEPARTURE of job J1.
StdOutSimEntityListener t=172.0, queue=Q2: DEPARTURE of job J2.
StdOutSimEntityListener t=183.0, queue=Q2: DEPARTURE of job J3.
StdOutSimEntityListener t=194.0, queue=Q2: DEPARTURE of job J4.
StdOutSimEntityListener t=205.0, queue=Q2: DEPARTURE of job J5.
```

In the example, 5 jobs are scheduled to arrive at a tandem queueing system consisting of the sequence of two `P_LCFS` queues named `Q1` and `Q2`. The jobs arrive at $t = 1, 2, \ldots, 5$, and

require service times at *each* of the `P_LCFS` queues of 10, 20, ..., 50, respectively. Given the high required service times compared to the interarrival times, the jobs leave `Q1`, in reverse order of arrival. However, again because of the relatively high required service times, no job can depart from `Q2` before the next arrival at that queue. As a result, `Q2` again reverses the order of arriving job in its departure process, and jobs depart from `Q2` in order of arrival at the first queue (and at the virtual tandem queue).

Despite the intriguing result that a tandem queue consisting of two `P_LCFS` queues *can* behaves like a `FCFS` queue with modified service-time requirement, and the fact that the tandem queue was so easy to realize, we want to focus at the implementation of the tandem queue, because there are some issues with it. Most important, we explained in the beginning of this section, that a composite queue behaves like the equivalent of its subqueues equipped with some *rouing* of jobs between these subqueues. However, in the example, there is no notion of a composite `SimQueue` at all! The job-arrival process "somehow know" that `Q1` is the first queue at which jobs must arrive, and the jobs *themselves* route themselves to the "next queue". In other words, all the "logic" related to routing jobs in a tandem queue is implemented in the arrival process and in the job implementation, whereas we really want these to be part of the tandem queue, the `SimQueue`. This, for instance, would allow *any* to properly visit the composite `SimQueue` without knowledge on its (internal) structure.

Well, allowing *any* `SimJob` to visit the tandem queue is not that difficult, as is shown in Listing 8.7 below. All we have to do is listen to departure events on `Q1` and then schedule an arrival event at `Q2`.

Listing 8.7: A tandem queue consisting of two P_LCFS queues (using SimQueue).

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  final P_LCFS lcfs_1 = new P_LCFS (el, null);
  lcfs_1.setName ("Q1");
  final P_LCFS lcfs_2 = new P_LCFS (el, null);
  lcfs_2.setName ("Q2");
  lcfs_1.registerSimEntityListener (new DefaultSimEntityListener ()
  {
    @Override
    public void notifyDeparture (final double time, final SimJob job, final SimQueue queue)
    {
      el.schedule (time, (SimEventAction) (SimEvent event) ->
      {
        lcfs_2.arrive (time, job);
      });
    }
  });
  for (int j = 1; j <= 5; j++)
  {
    final SimJob job = new DefaultSimJob (el, "J" + j, 10 * j);
    lcfs_1.scheduleJobArrival (j, job);
    job.registerSimEntityListener (new StdOutSimEntityListener ());
  }
  el.run ();
}
```

The result of the program is identical to that shown in 8.6, and the program itself is surprisingly short (we no longer have to create a dedicated `SimJob` for the tandem queue). This suggests (luckily) that the `jqueues` package has sufficient means to model composite queues manually. However, we are still faced with the problems that there is not really a notion of a composite queue, and we have not tackled the more generic problem of putting `SimQueues` is tandem:

- What if any of the subqueues *drops* the job?
- How to revoke a jobs from a composite queue?
- How to obtain statistics on the composite queue?

In other words, what we really want is an implementation of a `SimQueue` that behaves as the concatenation of visits to its subqueues. In Listing 8.8 we show this approach using a `BlackTandemSimQueue` described in this chapter, with its output in 8.9.

Listing 8.8: A tandem queue consisting of two P_LCFS queues (using BlackTandemSimQueue).

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  final P_LCFS lcfs_1 = new P_LCFS (el, null);
  lcfs_1.setName ("Q1");
  final P_LCFS lcfs_2 = new P_LCFS (el, null);
  lcfs_2.setName ("Q2");
  final Set<SimQueue> subQueues = new LinkedHashSet<>  ();
  subQueues.add (lcfs_1);
  subQueues.add (lcfs_2);
  final BlackTandemSimQueue compositeQueue = new BlackTandemSimQueue (el, subQueues, null);
  for (int j = 1; j <= 5; j++)
  {
    final SimJob job = new DefaultSimJob (el, "J" + j, 10 * j);
    compositeQueue.scheduleJobArrival (j, job);
    job.registerSimEntityListener (new StdOutSimEntityListener ());
  }
  el.run ();
}
```

Listing 8.9: Output of the program in Listing 8.8.

```
StdOutSimEntityListener t=1.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J1.
StdOutSimEntityListener t=1.0, queue=Tandem[Q1,Q2]: START of job J1.
StdOutSimEntityListener t=2.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J2.
StdOutSimEntityListener t=2.0, queue=Tandem[Q1,Q2]: START of job J2.
StdOutSimEntityListener t=3.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J3.
StdOutSimEntityListener t=3.0, queue=Tandem[Q1,Q2]: START of job J3.
StdOutSimEntityListener t=4.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J4.
StdOutSimEntityListener t=4.0, queue=Tandem[Q1,Q2]: START of job J4.
StdOutSimEntityListener t=5.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J5.
StdOutSimEntityListener t=5.0, queue=Tandem[Q1,Q2]: START of job J5.
StdOutSimEntityListener t=161.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J1.
StdOutSimEntityListener t=172.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J2.
StdOutSimEntityListener t=183.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J3.
StdOutSimEntityListener t=194.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J4.
StdOutSimEntityListener t=205.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J5.
```

The latter example shows exactly what we want: The creation of a new `SimQueue` that behaves exactly like a tandem configuration of two other queues. Admittedly, the code in the example is not particularly smaller than that of the previous examples, and the latter examples can certainly be used as a starting point for the study of tandem (and other composite) queues. However, if, for instance, you want to create a new `SimQueue` type that is the equivalent of other queues that are somehow interconnected, or if you want to study nested composite queues, then constructing a composite queue is a much faster (and more reliable) approach.

The extensive support for composite queues is one of the most distinguishing features of `jqueues`. Next to tandem queues, there is support for various types of *feedback* queues, *encapsulated* queues and *parallel* queues. In the next chapter we describe these composite queues in detail. In the remainder of the present chapter, we introduce some important concepts that apply to *any* composite queue.

8.2 The SimQueueComposite Interface

In the `jqueues` implementation, composite queues implement the `SimQueueComposite` interface, which, in turn, inherits from `SimQueue`.

SimQueueComposite			
Super			
`SimQueue`	See Section ??.		
Essential Properties			
`Queues`	C	`Set<SimQueue>`	The sub-queues; non-`null` and non-empty.
`SimQueueSelector`	C	`SimQueueSelector`	The sub-queue selector.
`DelegateSimJobFactory`	C	`DelegateSimJobFactory`	The factory for delegate jobs.
`StartModel`	V	`StartModel`	The start model (virtual).

8.2.1 The 'Queues' Property

The value of the 'Queues' property of a `SimQueueComposite` is a `Set` holding the subqueues of the composite queue. The property can only be set upon construction of the composite queue; and cannot be changed thereafter, but the sub-queues can be accessed through `getQueues ()` ↪ . (Note that the `Set` returned should *not* be modified.) Since the order of the subqueues is important to most composite-queue types, the implementation of `getQueues ()` *must* maintain a deterministic ordering of the sub-queues.

8.2.2 The 'SimQueueSelector' Property and Type

The `SimQueueSelector` property of a `SimQueueComposite` is of type `SimQueueSelector`, an interface capable of routing jobs through queues, in this particular case, of routing *delegate* jobs through *sub-queues*. The property is set (or created) upon construction and cannot be changed afterwards. Note that for most concrete implementations of `SimQueueComposite`, the route of jobs through the sequence of sub-queues is implied, and the queue selector is simply created on the fly in the constructor of the `SimQueueComposite`.

SimQueueSelector	
Methods	
`void resetSimQueueSelector ()`	Resets the selector.
`DQ selectFirstQueue ()`	Selects the sub-queue to visit first.
`DQ selectNextQueue (DQ)`	Selects the next sub-queue to visit.

The `SimQueueSelector` of a `SimQueueComposite` can have a *state* in the same sense as for `SimQueues`; the selector is therefore automatically *reset* upon a reset at the `SimQueueComposite` to which it is attached. (The reverse, by the way, is not true in general.)

A `SimQueueComposite` *must* always consult its `SimQueueSelector` in order to find the first or next sub-queue to visit, even if that is obvious from the definition of the composite queue (like in, for instance, `Tandem` described in Section 8.3.1). If either of the selection methods return `null`, the real job by default departs from the `SimQueueComposite`, though there are exceptions to this rule (like in the `Col` queueing system described in Section 8.8.1).

8.2.3 The 'DelegateSimJobFactory' Property and Type

The value of the `DelegateSimJobFactory` property is an object of type `DelegateSimJobFactory` ↪ , i.e., capable of creating *delegate* jobs from *real* jobs. It is passed or created upon construction of the `SimQueueComposite`. It can be changed afterwards at any time.

DelegateSimJobFactory	
Annotations	
`@FunctionalInterface`	
Methods	
`void resetFactory (double, Q)`	Resets the factory.
`DJ newInstance (double, J, Q)`	Creates a new delegate job.

The `double` arguments hold the (reset/current) time.

8.2.4 The 'StartModel' Hidden Property and Type

The `StartModel` property of a `SimQueueComposite` is *hidden*, see Section **??**, and *essential*, see Section **??**. Nonetheless, it is crucial in quickly understanding and referring to the mapping of the life-cycle of a real job on a specific type of composite queue onto that of the corresponding delegate job, and vice versa. It also describes the semantics of the server-access credits on the composite queue. We will describe the possible values of this property (up to now) in the next sections.

`LocalStart`

In the `LocalStart` model, arriving real jobs at the composite queue have to acquire a server-access credit on that queue in order for their delegate jobs to arrive at their respective sub-queues. Once a credit is obtained, the corresponding delegate job arrives at the first sub-queue, and the real job enters the service area of the composite queue. When a delegate job exits in any other way than through departure from a sub-queue, the corresponding real job exist the composite queue in the same way (i.e., drop or revocation; auto-revocations on any of the sub-queues are *not* allowed). When it departs from a sub-queue, the composite queue invokes the `selectNextQueue` on its `SimQueueSelector`, and lets the delegate jobs arrive at its new destination, or, if the method returns `null`, discards the delegate job and lets the real job depart from the composite queue. In other words, routing the delegate job through the chain of sub-queues takes place while the real job is in the service area.

`CTandem2`

In the `CTandem2` model, the composite queue has exactly two (obviously, distinct) sub-queues, named, respectively, the *wait* and *serve* queues. If a real job is present on the composite queue, then so is its delegate job on one of the two sub-queues; if the real job is in the waiting area, then the delegate job resides on the wait queue, if the real job is in the service area, then the delegate job resides on the serve queue. In other words, the start of the real job coincides with the transfer of the delegate job from the wait to the serve queue.

`Encapsulator`

In the `Encapsulator` model, the composite queue has a single sub-queue, named, *encapsulated* queues. If a real job is present on the composite queue, then so is its delegate job on the sub-queue. By default, if the real job is in the waiting area, then the delegate job resides in the waiting area of the encapsulated queue, if the real job is in the service area, then the delegate job resides in the serve area of the encapsulated queue. The server-access credits on both (composite and encapsulated) queues are synchronized.

8.3 Tandem Queues

In tandem queues, visiting jobs must visit each sub-queue once in a predetermined (and fixed) sequence. The list of (distinct) sub-queues is passed upon construction, and cannot be changed afterwards.

8.3.1 The `Tandem SimQueue`

Tandem	
Description	Serves jobs by letting their delegate jobs visit the sequence of sub-queues $\mathcal{Q}$ exactly once.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipline.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`Name`	The name, default "Tandem[$\mathcal{Q}$]"; non-`null`; RW.

8.3.2 The `CTandem2` `SimQueueComposite`

CTandem2 Compressed Tandem Queue with two Sub-Queues	
Description	A two sub-queue tandem queue that lets jobs wait in the waiting area of the first queue, the waiting queue, and serves them according to the service area of the second queue.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`Name`	The name, default "CT2$[Q_w, Q_s]$"; non-`null`; non-`null`; RW.
`StartModel`	The start model; `CTandem2`; RO.
`EncQueue`	The encapsulated queue Q_e (only sub-queue); RO.

8.4 Parallel Queues

In *paralllel* queueus, arriving jobs are sent to one and exactly one of the sub-queues, at which they are served until completion (exit), at which moment the real job exits the composite queue. Parallel queues only differ in their assignment strategy of an arriving job to the sub-queue at which the job is to be served.

8.4.1 The `JSQ` `SimQueueComposite`

JSQ Join Shortest Queue	
Description	Serves jobs until exit on the sub-queue in $\mathcal{Q}$ with the smallest number of jobs present (in service). Ties are broken at random with equal probabilities.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`Name`	The name, default "JSQ[$\mathcal{Q}$]"; non-`null`; RW.

8.4.2 The `JRQ SimQueueComposite`

JRQ Join Random Queue	
Description	Serves jobs until exit on a random sub-queue in $\mathcal{Q}$.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`Name`	The name, default "JRQ[$\mathcal{Q}$]"; non-`null`; RW.

8.4.3 The `Pattern SimQueueComposite`

Pattern	
Pattern	
Description	Serves jobs until exit on the sub-queue in $\mathcal{Q}$ assigned according to a predefined pattern.
Super	
`SimQueue`	See Section **??**.
`SimQueueComposite`	See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`Pattern`	The assignment pattern; `int[]`; RO.
`Name`	The name, default "Pattern[$\mathcal{Q}$]"; non-`null`; RW.

8.4.4 The `Parallel SimQueueComposite`

Par Parallel	
Description	Serves jobs until exit on the sub-queue in $\mathcal{Q}$ selected by a user-supplied selector.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`Pattern`	The assignment pattern; `int[]`; RO.
`Name`	The name, default "Par[$\mathcal{Q}$]"; non-`null`; RW.

8.5 Feedback Queues

An *feeback* queue has exactly one sub-queue which may be visited more than once.

8.5.1 The `FB_v SimQueueComposite`

FB_v Feedback Queue with Fixed Number of Visits.	
Description	Serves jobs until exit a fixed number of times on the sub-queue in $\mathcal{Q}$.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`NumberOfVists`	The number of visits; > 0; RO.
`Name`	The name, default "FB_{NumberOfVisits}[EncQueue]"; non-`null`; RW.

8.5.2 The `FB_p` `SimQueueComposite`

FB_p Probabilistic Feedback Queue	
Description	Serves jobs on the sub-queue in $\mathcal{Q}$, and upon exit at the sub-queue, feeds back back a job to the input with fixed probability, departing otherwise.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last sub-queue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`FeedbackProbability`	The feedback probability; ≥ 0; ≤ 1, RO.
`Name`	The name, default "FB_{FeedbackProbability}[EncQueue]"; non-`null`; RW.

8.5.3 The `FB` `SimQueueComposite`

FB General Feedback Queue.	
Description	Serves jobs on the sub-queue in $\mathcal{Q}$, and upon exit at the sub-queue, feeds back a job to the input or makes it depart from the composite queue under control of a user-supplied *controller*.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`FeedbackController`	The feedback controller; RO.
`Name`	The name, default "FB[EncQueue]"; non-`null`; RW.

8.6 Jackson Queues

8.6.1 The `Jackson` `SimQueueComposite`

Jackson Jackson Queueing System.	
Description	Routes jobs between sub-queues or departure according to a fixed routing probability matrix.
Super	
`SimQueue` `SimQueueComposite`	See Section ??. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`PdfArrival`	The arrival pdf; RO.
`PdfTransition`	The transition pdf; RO.
`Name`	The name, default "Jackson[$\mathcal{Q}$]"; non-`null`; RW.

8.7 Encapsulator Queues

An *encapsulator* queue has exactly one subqueue.

8.7.1 The `Enc SimQueueComposite`

Enc Encapsulator	
Description	Mimics the behavior of its (single) sub-queue.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `Encapsulator`; RO.
`Name`	The name, default "Enc[EncQueue]"; non-`null`; RW.

8.7.2 The `EncHS` `SimQueueComposite`

EncHS Encapsulator-Hide-Start	
Description	Mimics the behavior of its (single) sub-queue, but hides the start of (delegate) jobs on the encapsulated queue.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
State Invariants	
$\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q\in\mathcal{Q}} \lvert J_q(t)\rvert$.	
Properties	
`StartModel`	The start model; `Encapsulator`; RO.
`Name`	The name, default "EncHS[EncQueue]"; non-`null`; RW.

8.7.3 The `EncTL` `SimQueueComposite`

EncTL Encapsulator-Time-Limit	
Description	Mimics the behavior of its (single) sub-queue, but respects fixed expiration times for waiting, service and sojourn times.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
QAV	Drops arriving jobs during QAVs.
NoWaitArmed	Always `true`.
Waiting Area	Infinite waiting area, FIFO discipine.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `Encapsulator`; RO.
`MaxWaitingTime`	The maximum waiting time T_{wa}; RO.
`MaxServiceTime`	The maximum service time T_{se}; RO.
`MaxSojournTime`	The maximum sojourn time T_{so}; RO.
`Name`	The name, default "EncTL(T_{wa}, T_{se}, T_{so})[EncQueue]"; non-`null`; RW.

8.7.4 The `EncJL` `SimQueueComposite`

EncJL Encapsulator-Job-Limit	
Description	Mimics the behavior of its (single) sub-queue, but respects fixed limits on the number of jobs in the waiting area, the service area, and the queue as a whole.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `Encapsulator`; RO.
`MaxJobsInWaitingArea`	The maximum number L_{wa} of jobs in the waiting area; RO.
`MaxJobsInServiceArea`	The maximum number L_{se} of jobs in the service area; RO.
`MaxJobs`	The maximum number L_{so} of jobs in the queueing system; RO.
`Name`	The name, default "EncJL(L_{wa}, L_{se}, L_{so})[EncQueue]"; non-`null`; RW.

8.7.5 The `EncXM` `SimQueueComposite`

EncXM Encapsulator-Exit-Mapper	
Description	Mimics the behavior of its (single) sub-queue, but transforms a job exit on the sub-queue onto a possibly different exit method on the composite queue.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Set QAV	Ends/starts a QAV.
Arrival	The queue accepts arrivals.
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `Encapsulator`; RO.
`DropMapping`	The drop mapping; `MappableExitMethod`; RO.
`AutoRevocationMapping`	The auto-revocation mapping; `MappableExitMethod`; RO.
`DepartureMapping`	The departure mapping; `MappableExitMethod`; RO.
`Name`	The name, default "EncXM($\{X_a \to X_b\}$*])[EncQueue]"; non-`null`; RW.

8.8 Collector Queues

8.8.1 The `Col SimQueueComposite`

Col Collector	
Description	A queueing system with a *main* queue and a *collector* queue. Serves jobs on the main queue, but when it exits there, allows the job to be collected on the *collector* queue. Main and collector queues may be equal.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`MainQueue`	The main queue; RO.
`CollectorQueue`	The collector queue; RO.
`CollectDrops`	Whether to collect drops; RO.
`CollectAutoRevocations`	Whether to collect auto-revocations; RO.
`CollectDepartures`	Whether to collect departures; RO.
`DepartureMapping`	The departure mapping; `MappableExitMethod`; RO.
`Name`	The name, default "Col($\{X_a \to X_b\}$*])[MainQueue→CollectorQueue]"; non

8.8.2 The `DropCol SimQueueComposite`

DropCol Drop Collector	
Description	A queueing system with a *main* queue and a *collector* queue, the *drop* queue. Serves jobs on the main queue, but when dropped there, sends them to the collector queue. Main and collector queues may be equal.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any subqueue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`MainQueue`	The main queue; RO.
`CollectorQueue`	The collector queue; RO.
`DropQueue`	`== CollectorQueue`.
`Name`	The name, default "DropCol[MainQueue→CollectorQueue]"; non-`null`; RW.

8.8.3 The `ARevCol` `SimQueueComposite`

ARevCol Auto-Revocation Collector	
Description	A queueing system with a *main* queue and a *collector* queue, the *auto-revocati* Serves jobs on the main queue, but when auto-revoked there, sends them to the collector queue. Main and collector queues may be equal.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any sub-queue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`MainQueue`	The main queue; RO.
`CollectorQueue`	The collector queue; RO.
`AutoRevocationQueue`	`==CollectorQueue`.
`Name`	The name, default "ARevCol[MainQueue→CollectorQueue]"; non-`null`; RW.

8.8.4 The `DepCol` `SimQueueComposite`

DepCol Departure Collector	
Description	A queueing system with a *main* queue and a *collector* queue, the *departure* queue Serves jobs on the main queue, but when they depart there, sends them to the collector queue. Main and collector queues may be equal.
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any sub-queue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`MainQueue`	The main queue; RO.
`CollectorQueue`	The collector queue; RO.
`DepartureQueue`	`==CollectorQueue`.
`Name`	The name, default "DepCol[MainQueue→CollectorQueue]"; non-`null`; RW.

8.9 Generic Composite Queues

8.9.1 The `Comp_LS` `SimQueueComposite`

Comp_LS Generic Composite Queue with Local-Start Model	
Description	A queueing system with a `LocalStart` model and user-supplied queue selector (i.e., `SimQueueSelector`).
Super	
`SimQueue` `SimQueueComposite`	See Section **??**. See Section 8.2.
State	
NoWaitArmed	Always `true`.
SAC	The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue. If zero, arriving jobs wait in the waiting area. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Holds the jobs whose delegate job have arrived *anywhere*.
State Operations	
Drop	Drops arriving jobs during QAVs. Drops a visiting job if its delegate job is dropped on any sub-queue.
Revocation	Jobs can be revoked while waiting and while being served.
Set SAC	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs if SAC > 0.
Start	Starts a job if its delegate job arrives *anywhere*.
Departure	Jobs depart if its delegate job departs from the last subqueue.
Properties	
`StartModel`	The start model; `LocalStart`; RO.
`Name`	The name, default "Comp_LS[$\mathcal{Q}$]"; non-`null`; RW.

Chapter 9

Listeners

In `jqueues`, monitoring the progress of a running simulation, or perhaps calculating statistiscs on it, starts with chosing the proper *listeners*. During the simulation, queues and jobs, from hereon collectively referred to as *entities*, are obliged to notify registered listeners of (at least) *all* changes to their states. A listener is a `Java` object implementing the required methods allowing such notifications from the entity.

Since in most practical simulation studies, the ambition level is somewhat higher that showing events on `System.out`, we will delve somewhat deeper into listeners types and how to create, modify and register them.

In the example of Listing 3.1, we used a convenience method `registerStdOutSimQueueListener` ↪ to register a listener at the `FCFS_B` queue that simply writes the details of such notifications to the standard output, `Sytem.out` in `Java`. This is extremely handy for initial testing of a simulation, but in almost all cases, a more sophisticated listener is required; one you have to create yourself. Luckily, `jqueues` comes with a large collection of listener implementations, each for a specific purpose, that you can modify to suit your needs.

Restricting ourselves for the moment to queue listeners, we can create and register our own listener that reports to `System.out` in the code in Listing 3.1:

Listing 9.1: Creating and registering a listener.

```
final SimQueueListener listener = new StdOutSimQueueListener();
queue.registerSimEntityListener(listener);
```

Running the program of 3.1 again with this modified code fragment will yield (roughly) the same output, so we have not gained anything so far. However, a `StdOutSimQueueListener` allows all (notification) methods to be overridden, so we can, for instance, suppress certain notifications in the output like this:

In Listing 9.2, we modify the `StdOutSimQueueListener` by overriding the notification methods for server-access credits and queue-access vacations (which we do have not described yet), for the `StartArmed`-related notifications and replacing them with empty methods, effectively suppressing their respective output on `System.out`. In addition, we suppress the `UPDATE` and `STATE CHANGED` notifications. Our modified listener yields the following output:

If, on the other hand, your *only* interest is in the fundamental `RESET`, `UPDATE` and `STATE` ↪ `CHANGED` notifications, you can register a `StdOutSimEntityListener` as shown in Listing 9.4 or, simpler but equivalent, Listing 9.5, and their corresponding output in Listing 9.6.

In most practical cases, you will need a listener that does a bit more than reporting to `System` ↪ `.out`. Of course, you can override the methods in `StdOutSimQueueListener` to fit your purposes, but a better way is to use a `DefaultSimQueueListener`, or, if you just want to process the fundamental notification (`RESET`, `UPDATE` and `STATE CHANGED`), a `DefaultSimEntityListener`.

Listing 9.2: Suppressing specific notifications in a `StdOutSimQueueListener`.

```
final SimQueueListener listener = new StdOutSimQueueListener ()
{

  @Override
  public void notifyStateChanged (double time, SimEntity entity, List notifications) {}

  @Override
  public void notifyUpdate (double time, SimEntity entity) {}

  @Override
  public void notifyStartQueueAccessVacation (double time, SimQueue queue) {}

  @Override
  public void notifyStopQueueAccessVacation (double time, SimQueue queue) {}

  @Override
  public void notifyNewStartArmed (double time, SimQueue queue, boolean startArmed) {}

  @Override
  public void notifyOutOfServerAccessCredits (double time, SimQueue queue) {}

  @Override
  public void notifyRegainedServerAccessCredits (double time, SimQueue queue) {}

};
queue.registerSimEntityListener (listener);
```

Listing 9.3: Example output of Listing 3.1 with the modified listener of Listing 9.2

```
t=0.0, queue=FCFS_B[2]: ARRIVAL of job 0.
t=0.0, queue=FCFS_B[2]: START of job 0.
t=0.0, queue=FCFS_B[2]: DEPARTURE of job 0.
t=1.0, queue=FCFS_B[2]: ARRIVAL of job 1.
t=1.0, queue=FCFS_B[2]: START of job 1.
t=2.0, queue=FCFS_B[2]: ARRIVAL of job 2.
t=3.0, queue=FCFS_B[2]: ARRIVAL of job 3.
t=3.2, queue=FCFS_B[2]: DEPARTURE of job 1.
t=3.2, queue=FCFS_B[2]: START of job 2.
t=4.0, queue=FCFS_B[2]: ARRIVAL of job 4.
t=5.0, queue=FCFS_B[2]: ARRIVAL of job 5.
t=5.0, queue=FCFS_B[2]: DROP of job 5.
t=6.0, queue=FCFS_B[2]: ARRIVAL of job 6.
t=6.0, queue=FCFS_B[2]: DROP of job 6.
t=7.0, queue=FCFS_B[2]: ARRIVAL of job 7.
t=7.0, queue=FCFS_B[2]: DROP of job 7.
t=7.6000000000000005, queue=FCFS_B[2]: DEPARTURE of job 2.
t=7.6000000000000005, queue=FCFS_B[2]: START of job 3.
t=8.0, queue=FCFS_B[2]: ARRIVAL of job 8.
t=9.0, queue=FCFS_B[2]: ARRIVAL of job 9.
t=9.0, queue=FCFS_B[2]: DROP of job 9.
t=14.200000000000001, queue=FCFS_B[2]: DEPARTURE of job 3.
t=14.200000000000001, queue=FCFS_B[2]: START of job 4.
t=23.0, queue=FCFS_B[2]: DEPARTURE of job 4.
t=23.0, queue=FCFS_B[2]: START of job 8.
t=40.6, queue=FCFS_B[2]: DEPARTURE of job 8.
```

Listing 9.4: Creating and registering a `StdOutSimEntityListener`.

```
final SimEntityListener listener = new StdOutSimEntityListener ();
queue.registerSimEntityListener (listener);
```

Listing 9.5: Using `registerStdOutSimEntityListener`.

```
queue.registerStdOutSimEntityListener ();
```

Listing 9.6: Example output of Listing 3.1 with the modified listener of Listings 9.4 or 9.5

```
StdOutSimEntityListener t=0.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=0.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[0]@FCFS_B[2]]
  => START [Start[0]@FCFS_B[2]]
  => DEPARTURE [Dep[0]@FCFS_B[2]]
StdOutSimEntityListener t=1.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=1.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[1]@FCFS_B[2]]
  => START [Start[1]@FCFS_B[2]]
  => STA_FALSE [StartArmed[false]@FCFS_B[2]]
StdOutSimEntityListener t=2.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=2.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[2]@FCFS_B[2]]
StdOutSimEntityListener t=3.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=3.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[3]@FCFS_B[2]]
StdOutSimEntityListener t=3.2, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=3.2, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[1]@FCFS_B[2]]
  => START [Start[2]@FCFS_B[2]]
StdOutSimEntityListener t=4.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=4.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[4]@FCFS_B[2]]
StdOutSimEntityListener t=5.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=5.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[5]@FCFS_B[2]]
  => DROP [Drop[5]@FCFS_B[2]]
StdOutSimEntityListener t=6.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=6.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[6]@FCFS_B[2]]
  => DROP [Drop[6]@FCFS_B[2]]
StdOutSimEntityListener t=7.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=7.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[7]@FCFS_B[2]]
  => DROP [Drop[7]@FCFS_B[2]]
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[2]@FCFS_B[2]]
  => START [Start[3]@FCFS_B[2]]
StdOutSimEntityListener t=8.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=8.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[8]@FCFS_B[2]]
StdOutSimEntityListener t=9.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=9.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[9]@FCFS_B[2]]
  => DROP [Drop[9]@FCFS_B[2]]
StdOutSimEntityListener t=14.200000000000001, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=14.200000000000001, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[3]@FCFS_B[2]]
  => START [Start[4]@FCFS_B[2]]
StdOutSimEntityListener t=23.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=23.0, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[4]@FCFS_B[2]]
  => START [Start[8]@FCFS_B[2]]
StdOutSimEntityListener t=40.6, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=40.6, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[8]@FCFS_B[2]]
  => STA_TRUE [StartArmed[true]@FCFS_B[2]]
```

Both listener type are concrete, but all required method implementations are empty. In our next example, we take a `DefaultSimQueueListener` and modify it in order to calculate the average job *sojourn* time. This time, we create a separate `class` named `JobSojournTimeListener` for the listener, shown in Listing 9.7.

Listing 9.7: A (somewhat naive) queue listener that calculates the average job sojourn time.

```
public class JobSojournTimeListener
extends DefaultSimQueueListener
{

  private final Map<SimJob, Double> jobArrTimes = new HashMap<> ();

  private int jobsPassed = 0;
  private double cumJobSojournTime = 0;

  @Override
  public void notifyArrival (double time, SimJob job, SimQueue queue)
  {
    if (this.jobArrTimes.containsKey (job))
      throw new IllegalStateException ();
    this.jobArrTimes.put (job, time);
  }

  @Override
  public void notifyDeparture (double time, SimJob job, SimQueue queue)
  {
    if (! this.jobArrTimes.containsKey (job))
      throw new IllegalStateException ();
    final double jobSojournTime = time - this.jobArrTimes.get (job);
    if (jobSojournTime < 0)
      throw new IllegalStateException ();
    this.jobArrTimes.remove (job);
    this.jobsPassed++;
    this.cumJobSojournTime += jobSojournTime;
  }

  @Override
  public void notifyDrop (double time, SimJob job, SimQueue queue)
  {
    notifyDeparture (time, job, queue);
  }

  public double getAvgSojournTime ()
  {
    if (this.jobsPassed == 0)
      return Double.NaN;
    return this.cumJobSojournTime / this.jobsPassed;
  }

}
```

In the class, we override the (courtesy) notifications for job arrivals, departures and drops. When a job arrives, its arrival time is put into a private `Map` (`jobArrTimes`) for later reference. When a job departs or is dropped, we retrieve its arrival time, calculate its sojourn time, and add the result to the cumulative sum of sojourn times, `cumJobSojournTime`. In order to later interpret this value, we also have to maintain the number of jobs passed in a private field `jobsPassed`. At any time, the class provides the average sojourn time (over all jobs *passed*) through its `getAvgSojournTime` method. The calculation involved is trivial; the method returns `Double.NaN` when no jobs have passed.

The use of `JobSojournTimeListener` and the corresponding output are shown in Listings 9.8 and 9.9, respectively.

Before going into details on the average sojourn time reported, we want to stress that our implementation of `JobSojournTimeListener` is far from complete and even erroneous, although it works correctly in this specific (use) case. For instance, it fails to consider the fact that jobs may *not* leave the queue (in whatever way). Such jobs are named *sticky* jobs, and in a true application we would have to consider them. Second, apart from `DROP` and `DEPARTURE`, there are other means by which a job can depart the queueing system (in particular, *revocations*). Third, the listener ignores `RESET` notifications from the queue; a critical error as we shall see later. We will not further discuss these and other complications here, because our primary intention is to show you the mechanisms for creating and modifying listeners. We just want to point out that the design of *robust* statistical listeners is more complicated that shown here. We provide a thorough treatment in Chapter 10.

Listing 9.8: Our `FCFS_B` example with the custom `JobSojournTimeListener`.

```
final SimEventList el = new DefaultSimEventList ();
final int bufferSize = 2;
final FCFS_B queue = new FCFS_B (el, bufferSize);
final JobSojournTimeListener listener = new JobSojournTimeListener ();
queue.registerSimEntityListener (listener);
for (int j = 0; j < 10; j++)
{
  final double jobServiceTime = (double) 2.2 * j;
  final double jobArrivalTime = (double) j;
  final String jobName = Integer.toString (j);
  final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
  SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
el.run ();
System.out.println ("Average job sojourn time is " + listener.getAvgSojournTime () + ".");
```

Listing 9.9: The output of Listing 9.8.

```
Average job sojourn time is 7.06.
```

Returning to the reported average job sojourn time. Is it correct? Well, in order to verify, we have no choice but to carefully analyze the behavior of the `FCFS_B` queue under the given workload of jobs, as given in Table 9.1. The table shows for each job its job number (Job), arival time (Arr), required service time (ReQ), jobs waiting upon its arrival (WQA), start time (Start, if applicable), exit time (either due to departure or due to dropping), sojourn time (exit time minus arrival time), and remarks if applicable. The final rows show the sum (TOT) and the average (AVG) of the required service times and the sojourn times, thus validating the result.

Table 9.1: Analysis of job sojourn times in Listing 3.1.

Job	Arr	ReqS	WQA	Start	Exit	Sojourn	Remark
0	0.0	0.0	{}	0.0	0.0	0.0	Exits immediately.
1	1.0	2.2	{}	1.0	3.2	2.2	
2	2.0	4.4	{}	3.2	7.6	5.6	
3	3.0	6.6	{2}	7.6	14.2	11.2	
4	4.0	8.8	{3}	14.2	23.0	19.0	
5	5.0	11.0	{3, 4}	-	5.0	0.0	Dropped.
6	6.0	13.2	{3, 4}	-	6.0	0.0	Dropped.
7	7.0	15.4	{3, 4}	-	7.0	0.0	Dropped.
8	8.0	17.6	{4}	23.0	40.6	32.6	
9	9.0	19.8	{4, 8}	-	9.0	0.0	Dropped.
TOT		99.0				70.6	
AVG		9.9				7.06	

Chapter 10

Statistics

If you have read this book linearly up to this point, we hope you are wondering by now how to compute statistics like average job sojourn time at a queue, or average server utilization (just to name a few). Actually, this is somewhat on purpose: None of the interfaces and concrete implementations of jobs and queues provide direct support for maintaining and calculating statistics. This has three important reasons:

- It relieves the concrete `SimQueue` and `SimJob` implementations of the tedious and error-prone responsability of maintaining and calculting statistics; their basis functionality in terms of e.g. implementing the proper queueing discipline is complicated enough as it is.
- It forces the implementation of statistics in a generic way, e.g., applicable to *any* queue type.
- It allows for easy replacement of *all* statistics-gathering and maintenance code with that of a professional third-party statistics package like **XXX**.

Needless to say, `jqueues` has full support for the implementation of statistics, and actually, provides a few basic, but highly effective and extensible classes for statistics gathering itself.

10.1 Manual Construction of a `SimEntityListener`

The starting point for gathering statistics of a `SimEntity` is to register as a suitable `SimEntityListener` ↪ to it, and update statistics upon notifications from the entity, in particular update notifications and visit-related notifications like arrivals, drops and departures. Recall that a `SimEntity` is required to notify registered listeners *just before* it state changes through an update notification, allowing for easy maintenance of "time-average-type" statistics like the average number of jobs present at a queue. In addition (and often requiring a bit more work) the visit-related notifications allow for maintenance of "visit-related" statistics, like the average job sojourn time at a queue.

In the following sections, we will first explain how to obtain statistics for a `SimQueue` or `SimJob` manually, following the approach outline above. Subsequently, we will introduce a few statistics-related class that intend to make it easier to obtain such statistics.

10.1.1 Example 1: The Average Number of Jobs at a Queue

In this statistics-related example, we create a `SimQueue` and a workload consisting of arriving `SimJobs`. Our objective is to run the simulation, and calculate the average number of jobs at the queue. To make the example more interesting, we create two queues, a FCFS queue and a Preemptive (Resume) LCFS queue. We equip each with a listener that maintains the number of jobs

at the queue over time. As shown in Listing 10.1 below, we use a `DefaultSimEntityListener` as base classs, making it easier to ignore all the notifications we are not interested in. Note that we are silently assuming that the listener is registered at a `SimQueue`, and not at some other `SimEntity` type. In addition, we left out some sanity checking, e.g., on the `time` parameter (should not be in the past).

Listing 10.1: Simple listener for maintaining and calculating the average number of jobs at a SimQueue.

```
private final static class AvgJStatListener
extends DefaultSimEntityListener
{

  public AvgJStatListener (SimQueue queue)
  {
    notifyResetEntity (queue);
    queue.registerSimEntityListener (this);
  }

  private double tStart = Double.NEGATIVE_INFINITY;
  private double tLast = Double.NEGATIVE_INFINITY;
  private double cumJ = 0;

  @Override
  public void notifyResetEntity (SimEntity entity)
  {
    tStart = entity.getEventList ().getTime ();
    tLast = tStart;
    cumJ = 0;
  }

  @Override
  public void notifyUpdate (double time, SimEntity entity)
  {
    cumJ += ((SimQueue) entity).getNumberOfJobs () * (time - tLast);
    tLast = time;
  }

  public final double getStartTime ()
  {
    return this.tStart;
  }

  public final double getEndTime ()
  {
    return this.tLast;
  }

  public final double calculate ()
  {
    if (tLast > tStart)
    {
      if (! Double.isInfinite (tLast - tStart))
        return cumJ / (tLast - tStart);
      else
        return 0;
    }
    else
      return 0;
  }

}
```

Note that we are only interested in two notifications from the `SimQueue`, *reset* and *update* notifications. Because the number of jobs at a `SimQueue` is a *simple function*, we can easily integrate it over time by considering all points in time at which the function value *can* change, and cumulating the product of the interval since the last update with the current value. By definition, such points in time are the *updates* of a `SimQueue` and the constract of the latter mandates that each update results in a notification at its listeners *before* any changes have been applied to the queue. Needless to say, this requires substantial discipline for implementations of `SimQueue`, but we use it here to our advantage. In the end, all we need to do it divide the cumulated value with the total time interval, taking special care of the possibility that the interval is $+\infty$.

For the start of the interval, we use the reset notification from the queue. At each reset, we take the start time from the current time of the event list (note that we cannot take the time from the `SimQueue` directly), and reset our internal statistics. Note that we call `notifyResetEntity` directly from the constructor as well, because we cannot rely on receiving a reset between construction of the object and starting the simulation. Also note that we may assume that the number of jobs at a reset, as well as upon construction of a `SimQueue` is zero,

conform the contract of a `SimQueue`. However, unless the event list or the `SimQueue` attached to it is reset to a finite time value, we must use `Double.NEGATIVE_INFINITY` as time value. This explains the default values used in `AvgJStatListener`.

We list the example program using the `AvgJStatListener` and its output in Listings 10.2 and 10.3 below. Note the very important `el.reset (0)` line in the main program. We will later show the effect of leaving out or forgetting this statement. The correctness of the calculated values in the output can be assessed easily as is shown in the comments of the main program. For `P_LCFS`, the program outputs one of two distinct values for the average, both of which are actually correct, due to an ambiguity in the event scheduling. (Which we, in all honesty, overlooked while constructing the example.) This is also explained in the comments of the program.

Listing 10.2: Example program for calculating the average number of jobs at two queues.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF_" + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL_" + j, j));
  }
  final AvgJStatListener avgJStatListener_fcfs = new AvgJStatListener (fcfs);
  final AvgJStatListener avgJStatListener_lcfs = new AvgJStatListener (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("__Start_time:_" + avgJStatListener_fcfs.getStartTime () + ".");
  System.out.println ("__End_Time__:_" + avgJStatListener_fcfs.getEndTime () + ".");
  System.out.println ("__Average_number_of_jobs:_" + avgJStatListener_fcfs.calculate () + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("__Start_time:_" + avgJStatListener_lcfs.getStartTime () + ".");
  System.out.println ("__End_Time__:_" + avgJStatListener_lcfs.getEndTime () + ".");
  System.out.println ("__Average_number_of_jobs:_" + avgJStatListener_lcfs.calculate () + ".");
}
```

Listing 10.3: Output from the example program in Listing 10.2.

```
FCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Average number of jobs: 1.2727272727272727.
P_LCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Average number of jobs: 1.9090909090909092 [OR 2.727272727272727, see comments].
```

As mentioned earlier, it is essential to reset the event list to a finite time value before use. Below in Listing 10.4 we show the output of the program if we leave out the `el.reset (0)` statement, essentially starting the simulation and, more importantly, gathering the statistics at $t = -\infty$. The result is zero, because we are taking the average value of a function over an infinite interval, knowing that is is only non-zero over a finite interval. Despite that fact that for this particular case, we seem to get away with the infinite interval, and are able to return a sensible value for the average, this does not hold for the general case. (For instance, suppose we would have scheduled an arrival at $t = -\infty$, the program would still return the incorrect zero value for the average.) As a rule of thumb, the use of statistics involving infinite values should be avoided at all times, again stressing the importance of resetting the event list to a finite value before its use.

Listing 10.4: Output from the example program in Listing 10.2 without resetting the event list to zero (i.e., starting at $t = -\infty$).

```
FCFS:
  Start time: -Infinity.
  End Time  : 11.0.
  Average number of jobs: 0.0.
P_LCFS:
  Start time: -Infinity.
  End Time  : 11.0.
  Average number of jobs: 0.0.
```

10.1.2 Example 2: The Sojourn Times of a Job at Visited Queues

In our second example, our objective is to gather the per-queue average sojourn times of a *single* job. In addition, we want to solve this problem by listening to that particular `SimJob`, instead of listening to the `SimQueue`s it potentially visits. As we shall see, listening to `SimJob`s is a bit more involved, because, typically, `SimJob`s are not attached to an event list, and therefore require special attention to `RESET` events. Also, we want to show how to deal with so-called *sticky jobs*, i.e., jobs that never leave (in whatever way) a `SimQueue` during a visit.

In Listing 10.5, we show the code for a suitable listener to a single job that maintains the average sojourn time of the queue it visits. The implementation of `SimEventListResetListener`, immediately reveal an importance difference with listening to `SimQueue`s: A `SimJob` is not necessarily associated with an event list. This means that resets from the event list (used by the queues), as explained in Section **??**, may not reach the job and its listener. We therefore insist upon receiving the event list as an argument to the constructor and register as a `SimEventListResetListener`. We do not, however, do this in the constructor of the listeners, but rather in the `main` program shown in Listing 10.6.

Listing 10.5: Simple listener for maintaining and calculating the per-`SimQueue` average sojourn time for a single `SimJob`.

```
  private final static class AvgSojStatListener
  extends DefaultSimJobListener
  implements SimEventListResetListener
  {

    public AvgSojStatListener (final SimJob job)
    {
      this.job = job;
      notifyResetEntity (job);
    }

    private final SimJob job;

    private final Map<SimQueue, Map<Integer, Double>> visitsMap = new LinkedHashMap<> ();

    private double lastArrTime = Double.NaN;

    private final Map<SimQueue, Double> sojournTimeMap = new LinkedHashMap<> ();

    @Override
    public final void notifyEventListReset (final SimEventList eventList)
    {
      notifyResetEntity (null);
    }

    @Override
    public void notifyResetEntity (final SimEntity entity_dummy)
    {
      this.visitsMap.clear ();
      this.lastArrTime = Double.NaN;
      this.sojournTimeMap.clear ();
    }

    @Override
    public void notifyArrival (final double time, final SimJob job, final SimQueue queue)
    {
      if (! Double.isNaN (this.lastArrTime))
        throw new IllegalStateException ();
      this.lastArrTime = time;
    }

    @Override
    public void notifyDrop (final double time, final SimJob job, final SimQueue queue)
    {
      notifyDeparture (time, job, queue);
    }

    @Override
    public void notifyRevocation (final double time, final SimJob job, final SimQueue queue)
```

```
  {
    notifyDeparture (time, job, queue);
  }

  @Override
  public void notifyAutoRevocation (final double time, final SimJob job, final SimQueue queue)
  {
    notifyDeparture (time, job, queue);
  }

  @Override
  public void notifyDeparture (final double time, final SimJob job, final SimQueue queue)
  {
    if (Double.isNaN (this.lastArrTime))
      throw new IllegalStateException ();
    final double newSojTime = time - this.lastArrTime;
    if (! this.visitsMap.containsKey (queue))
    {
      this.visitsMap.put (queue, new HashMap<> ());
      this.visitsMap.get (queue).put (1, newSojTime);
    }
    else
    {
      final int oldVisits = this.visitsMap.get (queue).keySet ().iterator ().next ();
      final double oldCumSojJ = this.visitsMap.get (queue).get (oldVisits);
      this.visitsMap.get (queue).clear ();
      this.visitsMap.get (queue).put (oldVisits + 1, oldCumSojJ + newSojTime);
    }
    this.lastArrTime = Double.NaN;
  }

  private void calculate (final double time)
  {
    this.sojournTimeMap.clear ();
    for (final Map.Entry<SimQueue, Map<Integer, Double>> entry : this.visitsMap.entrySet ())
    {
      final SimQueue queue = entry.getKey ();
      int visits = entry.getValue ().keySet ().iterator ().next ();
      double cumSojJ = entry.getValue ().get (visits);
      if ((! Double.isNaN (this.lastArrTime))
      &&  this.job.getQueue () == queue)
      {
        visits++;
        cumSojJ += (time - this.lastArrTime);
      }
      this.sojournTimeMap.put (entry.getKey (), cumSojJ / visits);
    }
    if ((! Double.isNaN (this.lastArrTime))
    &&  (! this.sojournTimeMap.containsKey (this.job.getQueue ())))
      this.sojournTimeMap.put (this.job.getQueue (), time - this.lastArrTime);
  }

  public final void report (final double time)
  {
    calculate (time);
    System.out.println ("Time = " + time + ":");
    if (this.sojournTimeMap.isEmpty ())
      System.out.println ("  No visits recorded!");
    else
      for (final Map.Entry<SimQueue, Double> entry : this.sojournTimeMap.entrySet ())
        System.out.println ("  Queue = " + entry.getKey ()
          + ", avg sojourn time = " + entry.getValue ());
  }

}
```

In the listener code, we maintain the per-`SimQueue` accumulated sojourn time of the `SimJob` in a `private` field named `visitsMap`. The additional level of indirection in the `Map` used holds the number of (completed) visits. Note that we listen to *all* event notifications relevant to the start of end of a visit, and that the `lastArrTime` is used to remember the start of the *current* visit of the job, if applicable. Special care is taken to ensure that if the job is *currently* visiting a `SimQueue` at the time of calculation, that particular visit is included in the calculation. (This, by the way, is merely a choice of definition of the average sojourn time. Note that you may equally well define it to *only* include *completed* visits. This is just to illustrate the importance of carefully defining your performance measures and statistics.)

Listing 10.6: Example program for calculating the average sojourn time of a single job visiting multiple queues.

```
  public static void main (final String[] args)
  {

    final SimEventList el = new DefaultSimEventList ();
    el.reset (0);

    final SimJob job = new DefaultSimJob (el, "job", 1.0);
    final AvgSojStatListener statListener = new AvgSojStatListener (job);
    job.registerSimEntityListener (statListener);
    el.addListener (statListener);

    final FB_p fb_25 = new FB_p (el, new FCFS (el), 0.25, null, null);
    final FB_p fb_50 = new FB_p (el, new FCFS (el), 0.50, null, null);
```

```
    final FB_p fb_75 = new FB_p (el, new FCFS (el), 0.75, null, null);
    final SINK sink  = new SINK (el);

    final int FEEDBACK_TANDEM_VISITS = 10000;

    job.registerSimEntityListener (new DefaultSimJobListener ()
    {

      private int fbVisitCycles = 0;

      @Override
      public void notifyDrop (final double time, final SimJob job, final SimQueue queue)
      {
        notifyDeparture (time, job, queue);
      }

      @Override
      public void notifyRevocation (final double time, final SimJob job, final SimQueue queue)
      {
        notifyDeparture (time, job, queue);
      }

      @Override
      public void notifyAutoRevocation (final double time, final SimJob job, final SimQueue queue)
      {
        notifyDeparture (time, job, queue);
      }

      @Override
      public void notifyDeparture (final double time, final SimJob job, final SimQueue queue)
      {
        if (queue == fb_25)
          SimJQEventScheduler.scheduleJobArrival (job, fb_50, time);
        else if (queue == fb_50)
          SimJQEventScheduler.scheduleJobArrival (job, fb_75, time);
        else if (queue == fb_75)
        {
          if (this.fbVisitCycles++ < FEEDBACK_TANDEM_VISITS)
            SimJQEventScheduler.scheduleJobArrival (job, fb_25, time);
          else
            SimJQEventScheduler.scheduleJobArrival (job, sink, time);
        }
        else
          throw new IllegalStateException ();
      }
    });

    fb_25.scheduleJobArrival (0, job);

    el.run ();
    statListener.report (el.getTime ());

    el.runUntil (el.getTime () + 1000, true, true);
    statListener.report (el.getTime ());

    el.reset ();
    statListener.report (el.getTime ());

  }
```

In the `main` code, a single `SimEventList` and `SimJob` are created along with an instance of the listener `AvgSojStatListener`. Then, the listener is attached to the `SimJob`, and registered as a `SimEventResetListener` at the `SimEventList`. Through a dedicated listener on it, the `SimJob` is then routed a fixed number of times (`FEEDBACK_TANDEM_VISITS`) along three probabilistic feedback queues with 25%, 50% and, 75% feedback probabilities, respectively, after which it enters a `SINK` queue.

The output of the program is shown in Listing **??**. Given the probabilistic nature of the `FB_p` queue, it is likely that you obtain different (though "close") results.

Listing 10.7: Output from the example program in Listing 10.6 (results are not deterministic!).

```
Time = 73063.0:
  Queue = FB_25.0%[FCFS], avg sojourn time = 1.337866213378662
  Queue = FB_50.0%[FCFS], avg sojourn time = 1.999000099990001
  Queue = FB_75.0%[FCFS], avg sojourn time = 3.9687031296870314
  Queue = SINK, avg sojourn time = 0.0
Time = 74063.0:
  Queue = FB_25.0%[FCFS], avg sojourn time = 1.337866213378662
  Queue = FB_50.0%[FCFS], avg sojourn time = 1.999000099990001
  Queue = FB_75.0%[FCFS], avg sojourn time = 3.9687031296870314
  Queue = SINK, avg sojourn time = 1000.0
Time = -Infinity:
  No visits recorded!
```

As expected, the average sojourn time on *any* of the `FB_p` queue with feedback probability p is close to

$$\frac{S_j}{1-p} = \frac{1}{1-p},$$

where s_j is the requested service time of the job j, which in this case is unity throughout. Hence, for the `FB_25%[FCFS]`, `FB_50%[FCFS]`, and `FB_75%[FCFS]` queues, the expected average job sojourn times amount to 4/3, 2, and 4, respectively.

10.1.3 Summary

From the two examples presented and analyzed in the preceding section, we summarize the approach towards gathering statistics through manually- created listeners:

- Precisely define the statistic you're interested in. Take special notice of corner case like a reset and sticky jobs.
- Select which `SimEntity` to listen to.
- Select or construct a suitable listener for the statistic, and register as listener to the entity.
- Make sure you revise the reset strategy. Are the entities and listeners involved properly reset upon construction, or before the start of the simulation? Does the code properly handle event-list resets? What about independent `SimQueue` resets? Does the code deal properly with `SimJobs` that are *not* attached to the `SimEventList`?
- Run the simulation, typically through invoking `runUntil` on the `SimEventList`.
- Calculate the statistic(s) through a separate (say) `calculate` method. Usually, such method needs time argument.
- Report and/or store the calculated results.
- Reset, optionally, re-populate and re-run the event list (e.g., for *replications*[1]).

In the next sections, we describe some default implementations of listeners aimed at statistics gathering.

10.2 The AbstractSimQueueStat Base Class

The examples in the previous sections explained how to obtain statistics on `SimQueues` and `SimJobs`. The next sections in this chapter introduce a few classes that either directly provide a few of the most basic statistics or allow you to define your own statistics more easily.

The `AbstractSimQueueStat` is the (abstract) base class for all subsequent classes. Its functionality is comparable to `AvgJStatListener` introduced in Section 10.1.1, with a few new functions and increased robustness, and delegating the actual maintenance and calulation of the statistic(s) to only a few abstract methods. To illustrate its use, we show in Listings 10.8 and 10.9 the modified listener and main program, respectively, from the example in Section 10.1.1 now using `AbstractSimQueueStat`.

Listing 10.8: Using AbstractSimQueueStat for a listener for maintaining and calculating the average number of jobs at a SimQueue.

```
private static class AvgJStatListener
extends AbstractSimQueueStat
{

  public AvgJStatListener (SimQueue queue)
  {
    super (queue);
  }
```

[1]Such sophisticated methods and techniques in discrete-event simulation are beyond the scope of this book, and of the `jsimulation` and `jqueues` libraries

```
  private double cumJ = 0;

  private double avgJ = 0;

  @Override
  protected void resetStatistics ()
  {
    cumJ = 0;
    avgJ = 0;
  }

  @Override
  protected void updateStatistics (double time, double dt)
  {
    cumJ += getQueue ().getNumberOfJobs () * dt;
  }

  @Override
  protected void calculateStatistics (double startTime, double endTime)
  {
    if (startTime == endTime)
      return;
    if (! Double.isInfinite (endTime - startTime))
      avgJ = cumJ / (endTime - startTime);
    else
      avgJ = 0;
  }

  public double getAvgJ ()
  {
    calculate ();
    return this.avgJ;
  }

}
```

Listing 10.9: Example program for calculating the average number of jobs at two queues using AbstractSimQueueStat.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF " + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL " + j, j));
  }
  final AvgJStatListener avgJStatListener_fcfs = new AvgJStatListener (fcfs);
  final AvgJStatListener avgJStatListener_lcfs = new AvgJStatListener (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("  Start time: " + avgJStatListener_fcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_fcfs.getLastUpdateTime () + ".");
  System.out.println ("  Average number of jobs: " + avgJStatListener_fcfs.getAvgJ () + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("  Start time: " + avgJStatListener_lcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_lcfs.getLastUpdateTime () + ".");
  System.out.println ("  Average number of jobs: " + avgJStatListener_lcfs.getAvgJ () + ".");
}
```

Compared to the manual approach in Section 10.1.1, we highlight the following differences:

- In the listener, we only have to implement what to do upon a reset (`resetStatistics`) and upon an update (`updateStatistics`), and how to calculate the (internally stored) result (`calculateStatistics`); the base class takes care of registering at `SimQueue` and internally storing the times of the last reset (or the time at construction) and of the last update.

- In the concrete listener, we provide a method for access to the internally stored result, but that method automatically invokes the base class' `calculate ()` method first. The base class attempts to avoid unnecessary recalculations of the result in its subclass.

- Users of the concrete subclass have access to the methods `getStartTime` and `getLastUpdateTime` ↪ at all times. Upon calculation (without a time argument), the time of the last update determines the upper boundary of the measurement interval.

- Users may also invoke `calculate (double endTime)` which allows the extension of the measurement interval beyond the last update time. This, however, prevents subsequent update *before* the `endTime` provided, unless the object (or the queue, or the underlying event list) is reset. Nonetheless, the feature is handy because in many practical cases, one wants to take the time average over a fixed time interval, instead of a time interval of which the upper boundary is determined by the last update. Unfortunately, one cannot chose and `endTime` smaller than the last update time (at the expense of an exception thrown).

- Users can change the `SimQueue` from which the statistics are obtained through `setQueue` ↪ `(Q)`. Setting the queue will immediately reset the statistics object. As expected, the base class `AbstractSimQueueStat` takes care of registering at the new queue (if any) after unregistering at the old queue. The only sensible use for that feature that we can think of is delayed initialization.

10.3 The SimpleSimQueueStat Class

The `SimpleSimQueueStat` class implements `AbstractSimQueueStat` described in Section 10.2 for some important statistics on a `SimQueue`:

- The average number of jobs;
- The average number of jobs in the service area;
- The maximum and minimum number of jobs;
- The maximum and minimum number of jobs in the service area.

Attaching a `SimpleSimQueueStat` to a `SimQueue` is a very conventient way of obtaining a first impression of the performance of the queue; we have used it a lot ourselves during testing. Below in Listing 10.10 we illustrate its use for the example in Section 10.1.1; the output (still ambiguous!) is shown in Listing 10.11. Note that the number of updates is also available from `SimpleSimQueueStat`, which is very handy for detailed update-related debugging on `SimQueue` implementations.

Listing 10.10: Example program for obtaing basic statistics using SimpleSimQueueStat.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
```

```
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF_" + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL_" + j, j));
  }
  final SimpleSimQueueStat avgJStatListener_fcfs = new SimpleSimQueueStat (fcfs);
  final SimpleSimQueueStat avgJStatListener_lcfs = new SimpleSimQueueStat (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("__Start_time:_" + avgJStatListener_fcfs.getStartTime () + ".");
  System.out.println ("__End_Time__:_" + avgJStatListener_fcfs.getLastUpdateTime () + ".");
  System.out.println ("__Number_of_updates_____________________:_"
    + avgJStatListener_fcfs.getNumberOfUpdates () + ".");
  System.out.println ("__Average_number_of_jobs________________:_"
    + avgJStatListener_fcfs.getAvgNrOfJobs () + ".");
  System.out.println ("__Average_number_of_jobs_in_service_area:_"
    + avgJStatListener_fcfs.getAvgNrOfJobsInServiceArea () + ".");
  System.out.println ("__Maximum_number_of_jobs________________:_"
    + avgJStatListener_fcfs.getMaxNrOfJobs () + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("__Start_time:_" + avgJStatListener_lcfs.getStartTime () + ".");
  System.out.println ("__End_Time__:_" + avgJStatListener_lcfs.getLastUpdateTime () + ".");
  System.out.println ("__Number_of_updates_____________________:_"
    + avgJStatListener_fcfs.getNumberOfUpdates () + ".");
  System.out.println ("__Average_number_of_jobs________________:_"
    + avgJStatListener_lcfs.getAvgNrOfJobs () + ".");
  System.out.println ("__Average_number_of_jobs_in_service_area:_"
    + avgJStatListener_lcfs.getAvgNrOfJobsInServiceArea ()+ ".");
  System.out.println ("__Maximum_number_of_jobs________________:_"
    + avgJStatListener_lcfs.getMaxNrOfJobs () + ".");
}
```

Listing 10.11: Output from the example program in Listing 10.10.

```
FCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Number of updates                    : 6.
  Average number of jobs               : 1.2727272727272727.
  Average number of jobs in service area: 0.9090909090909091.
  Maximum number of jobs               : 2.0.
P_LCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Number of updates                    : 6.
  Average number of jobs               : 1.9090909090909092 [OR 2.727272727272727, see comments].
  Average number of jobs in service area: 1.9090909090909092 [OR 2.727272727272727, see comments].
  Maximum number of jobs               : 3.0 [OR 4.0, see comments].
```

Note the remarkable difference in the average number of jobs in the service area between `FCFS` and `P_LCFS`. In the former, jobs waiting are only admitted to the service area if the previously arrived job has departed, whereas in the latter, jobs are admitted immediately to the service area upon arrival (because their service starts immediately), and preempted jobs are *not* transferred back into the waiting area. This illustrates the (surprising) distinction between the number of jobs *in service* (a concept not directly supported by `SimQueue`) and the number of jobs *in the service area* (in which service is allowed, but not guaranteed).

10.4 The AutoSimQueueStat Class

10.4.1 Introduction

The `SimpleSimQueueStat` class described in Section 10.3 provides a convenient way of obtaining some important statistics related to the number of jobs at a queue or at its service area. Although it is fairly easy to extend `SimpleSimQueueStat`, or even `AbstractSimQueueStat` for that matter, for other time-dependent performance measures, we quickly realize that the resulting code would only differ in *which* statistic we take from the queue upon updates. The `AutoSimQueueStat` class introduced in this section honors this observation, and relies on a so-called `SimQueueProbe` to obtain the momentary value of a specific performance measure at a `SimQueue`. It then automatically maintains the average, maximum and minimum values for all registered probes.

10.4.2 The SimQueueProbe Interface

As described in the previous section, the `SimQueueProbe` interface features obtaining the momentary value of a specific performance measure at a queue (the queue is actually a parameter,

so probes can be reused). Its interface definition is shown in Listing 10.12. Note that only double values (or values that can be cast to double) are supported.

Listing 10.12: The SimQueueProbe interface.

```
public interface SimQueueProbe<Q extends SimQueue>
{

  public double get (Q queue);

}
```

10.4.3 The AutoSimQueueStatEntry Class

The `AutoSimQueueStatEntry` class connects a `SimQueueProbe` with a name, and adds statistics maintenance for the values the probe provides. Unfortunately, unlike a `SimQueueProbe`, you cannot reuse a `AutoSimQueueStatEntry` among different queues.

10.4.4 Example

Below in Listing 10.13 we illustrate the use of `AutoSimQueueStat` for the example in Section 10.1.1; the output (still ambiguous!) is shown in Listing 10.14. Note that two probes are created, one for the number of jobs at a queue, and one for the number of jobs in the service area at a queue. For each queue, `FCFS` and `P_LCFS`, an array of entries is created, holding unique entries for the statistics required. Finally, note that `AutoSimQueueStat` features a method `report ()` (optionally, with an integer indentation parameter) for quickly reporting the calculated statistics for all registered probes.

Listing 10.13: Example program for obtaing statistics using AutoSimQueueStat.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF_" + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL_" + j, j));
  }
  final SimQueueProbe probeJ  =
    (SimQueueProbe) (SimQueue queue) -> queue.getNumberOfJobs ();
  final SimQueueProbe probeJX =
    (SimQueueProbe) (SimQueue queue) -> queue.getNumberOfJobsInServiceArea ();
  final List<AutoSimQueueStatEntry> entries_fcfs = new ArrayList<> ();
  final List<AutoSimQueueStatEntry> entries_lcfs = new ArrayList<> ();
  entries_fcfs.add (new AutoSimQueueStatEntry ("J",  probeJ));
  entries_fcfs.add (new AutoSimQueueStatEntry ("JX", probeJX));
  entries_lcfs.add (new AutoSimQueueStatEntry ("J",  probeJ));
  entries_lcfs.add (new AutoSimQueueStatEntry ("JX", probeJX));
  final AutoSimQueueStat stat_fcfs = new AutoSimQueueStat (fcfs, entries_fcfs);
  final AutoSimQueueStat stat_lcfs = new AutoSimQueueStat (lcfs, entries_lcfs);
  el.run ();
  System.out.println ("FCFS:");
  stat_fcfs.report (2);
  System.out.println ("P_LCFS:");
  stat_lcfs.report (2);
}
```

Listing 10.14: Output from the example program in Listing 10.10.

```
FCFS:
```

```
  Average J: 1.2727272727272727.
  Minimum J: 0.0.
  Maximum J: 2.0.
  Average JX: 0.9090909090909091.
  Minimum JX: 0.0.
  Maximum JX: 1.0.
P_LCFS:
  Average J: 1.9090909090909092 [OR 2.727272727272727, see comments].
  Minimum J: 0.0.
  Maximum J: 3.0 [OR 4.0, see comments].
  Average JX: 1.9090909090909092 [OR 2.727272727272727, see comments].
  Minimum JX: 0.0.
  Maximum JX: 3.0 [OR 4.0, see comments].
```

10.5 The SimpleSimQueueVisitsStat Class

The `SimpleSimQueueVisitsStat` maintains and calculates some important visits-related statistics on the queue it is registered at:

- The number of arrivals, of jobs that started and the number of departures;
- The number of dropped and (successfully) revoked jobs;
- The average waiting time (averaged over jobs that started);
- The average sojourn time (averaged over jobs that departed);
- The maximum and minimum waiting time (averaged over jobs that started);
- The maximum and minimum sojourn time (averaged over jobs that departed).

In Listing 10.15 below we provide an example of its use, again using the scheduling example from YYY; its corresponding output is shown in Listing 10.16.

Listing 10.15: Example program for obtaing basic statistics using SimpleSimQueueVisitsStat.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  // avgWaitJ = 0 + 0 + 0 + 1 + 3 / 5 = 0.8.
  // avgSojJ  = 0 + 1 + 2 + 4 + 7 / 5 = 14 / 5 = 2.8.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  // avgWaitJ = 0.
  // avgSojJ  = 0 + 1 + 9 + 7 + 4 / 5 = 21 / 5 = 4.2.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  // avgWaitJ = 0.
  // avgSojJ  = 0 + 10 + 9 + 7 + 4 / 5 = 30 / 5 = 6.0.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF " + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL " + j, j));
  }
  final SimpleSimQueueVisitsStat visitsStatListener_fcfs = new SimpleSimQueueVisitsStat (fcfs);
  final SimpleSimQueueVisitsStat visitsStatListener_lcfs = new SimpleSimQueueVisitsStat (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("  Start Time            : "
    + visitsStatListener_fcfs.getStartTime ()           + ".");
  System.out.println ("  End Time              : "
    + visitsStatListener_fcfs.getLastUpdateTime ()      + ".");
  System.out.println ("  Number of Arrivals    : "
    + visitsStatListener_fcfs.getNumberOfArrivals ()    + ".");
  System.out.println ("  Number of Started Jobs: "
    + visitsStatListener_fcfs.getNumberOfStartedJobs () + ".");
  System.out.println ("  Number of Departures  : "
    + visitsStatListener_fcfs.getNumberOfDepartures ()  + ".");
```

```
  System.out.println ("  Minimum Waiting Time  : "
    + visitsStatListener_fcfs.getMinWaitingTime ()      + ".");
  System.out.println ("  Maximum Waiting Time  : "
    + visitsStatListener_fcfs.getMaxWaitingTime ()      + ".");
  System.out.println ("  Average Waiting Time  : "
    + visitsStatListener_fcfs.getAvgWaitingTime ()      + ".");
  System.out.println ("  Minimum Sojourn Time  : "
    + visitsStatListener_fcfs.getMinSojournTime ()      + ".");
  System.out.println ("  Maximum Sojourn Time  : "
    + visitsStatListener_fcfs.getMaxSojournTime ()      + ".");
  System.out.println ("  Average Sojourn Time  : "
    + visitsStatListener_fcfs.getAvgSojournTime ()      + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("  Start Time           : "
    + visitsStatListener_lcfs.getStartTime ()           + ".");
  System.out.println ("  End Time             : "
    + visitsStatListener_lcfs.getLastUpdateTime ()      + ".");
  System.out.println ("  Number of Arrivals   : "
    + visitsStatListener_lcfs.getNumberOfArrivals ()    + ".");
  System.out.println ("  Number of Started Jobs: "
    + visitsStatListener_lcfs.getNumberOfStartedJobs () + ".");
  System.out.println ("  Number of Departures : "
    + visitsStatListener_lcfs.getNumberOfDepartures ()  + ".");
  System.out.println ("  Minimum Waiting Time  : "
    + visitsStatListener_lcfs.getMinWaitingTime ()      + ".");
  System.out.println ("  Maximum Waiting Time  : "
    + visitsStatListener_lcfs.getMaxWaitingTime ()      + ".");
  System.out.println ("  Average Waiting Time  : "
    + visitsStatListener_lcfs.getAvgWaitingTime ()      + ".");
  System.out.println ("  Minimum Sojourn Time  : "
    + visitsStatListener_lcfs.getMinSojournTime ()      + ".");
  System.out.println ("  Maximum Sojourn Time  : "
    + visitsStatListener_lcfs.getMaxSojournTime ()      + ".");
  System.out.println ("  Average Sojourn Time  : "
    + visitsStatListener_lcfs.getAvgSojournTime ()      + ".");
}
```

Listing 10.16: Output from the example program in Listing 10.15.

```
FCFS:
  Start Time           : 0.0.
  End Time             : 11.0.
  Number of Arrivals   : 5.
  Number of Started Jobs: 5.
  Number of Departures : 5.
  Minimum Waiting Time : 0.0.
  Maximum Waiting Time : 3.0.
  Average Waiting Time : 0.8.
  Minimum Sojourn Time : 0.0.
  Maximum Sojourn Time : 7.0.
  Average Sojourn Time : 2.8.
P_LCFS:
  Start Time           : 0.0.
  End Time             : 11.0.
  Number of Arrivals   : 5.
  Number of Started Jobs: 5.
  Number of Departures : 5.
  Minimum Waiting Time : 0.0.
  Maximum Waiting Time : 0.0.
  Average Waiting Time : 0.0.
  Minimum Sojourn Time : 0.0.
  Maximum Sojourn Time : 9.0. [OR 10.0, see comments].
  Average Sojourn Time : 4.2. [OR 6.0, see comments].
```

10.6 Conclusions

In this chapter, we have recommended means to obtain statistics from the entities (`SimEntitys`) in a discrete-event simulation (run). Even though `jqueueus` does not directly support statistics-gathering and analysis, it contains all the required hooks to that extent, either through the use of third-party libraries like the `SSJ` library from Université de Montréal `JSSL` [2], or through user-supplied code.

The starting point for obtaining statistics is the requirement that each `SimEntity` will report changes to its *event state* to all registered listeners. In addition, it will issue an `UPDATE` ↪ notification preceding the (possible) invocation of a method after a non-trivial amount of time since the current event state was obtained. These features allow for easy construction of statistics-gathering and analysis for statistics like average number of jobs visiting a queue, or average sojourn time at a queue. The `jqueues` packages contains a few classes for gathering and analyzing simple statistics.

Chapter 11

Epilogue

In this book, we describe two `Java` libraries for discrete-event simulation of queueing systems, (1) `jsimulation` provides support for events, event lists, event scheduling and actions, and (2) `jqueue` which extends `jsimulation` with support for the simulation of queueing systems. In Chapter 3 we provide a guided tour to both packages, whereas in Chapters 4 and 5 6 we describe in detail `jsimulation` and `jqueues`, respectively. In Chapters 7 and 8 we describe all available implementations of fundamental and composite queueing systems, respectively. In Chapter 9 we **XXX**. Finally, in Chapter 10, we explain how to obtain statistics from a simulation (run).

Although we have at this time of writing not yet completed the road map for our next release, we intend to include the following new features:

- The seeding of random-number generators upon a reset, either from a manually provided *seed value*, or from a suitable *seed generator*. The latter, though, requires a study on the implementation of `Java`'s `Random` class, i.e., its pseudo-random generator.
- A wider support for random-number generation, allowing other implementations than `Java`'s `Random` class.
- A new high-performance model for composite queues, either as a backwards-compatible extension to the present `SimQueueComposite` interface, or as a new queue type alongside it. The main idea is to allow jobs to visit sub-queues themselves, instead of through delegate jobs. We actually tried to include this feature into the current release, but this led to an impossibly complex concept of composite queues, and we did not manage to properly implement the current set of composite queues (and their associated tests). Therefore, we reverted our efforts, and satisfied ourselves with the current model for composite queues.
- Several single-server processor-sharing queueing disciplines in which the relative execution rates of the jobs in the service area depends on either:
  - The per-job QoS value, which we then need to restrict to a number. Our improvised name for this queueing system is `DPS`, or *Discriminatory Processor Sharing*.
  - The relative *share* of jobs with *identical* QoS values, i.e., (sorry for the confusion here) the per-job-*class* QoS value. This approach requires a `Map<P, Double>`, mapping QoS values (of generic type `P`) onto relative *per-job* execution rates. If applicable, each job present of a specific QoS value (i.e., class of jobs) receives the same relative priority. We intend to name this queueing system `PPS`, or *Priority Processor Sharing*.
  - The relative *share* of the job *class*. This approach is comparable to the previous one, yet makes multiple jobs of a single job class *share equally* their execution rate;

it also requires a `Map<P, Double>`, mapping QoS values (of generic type `P`) onto relative *per-class* execution rates. We intend to name this queueing system `GPPS`, or *Group-Priority Processor Sharing*.

More background on these queueing systems, and some motivation for their naming is given in [1].

- The `DUPS` (fundamental) queue type; the abbreviation stands for *Decay-Usage Processor Sharing*. In `DUPS`, jobs are served on a single processor-sharing queue, but their relative execution rates are inversely proportional to their obtained service times.

- Support for servers with different "capacity". For instance, we are considering the development of a `EncCap` encapsulator (composite) queue with an essential property `Capacity` that "scales" the required service time on the encapsulated queue. We are also considering an extension to the `SimJob` interface to include assessment of the required job service time in the presence of queues with varying capacity. At the moment, though, we do not consider server capacities that can *change* in time. (Since that would make the capacity a state property, and far more complicated.) Note that for certain queue types like server-less queues or `SUR`, the concept of server capacity has no meaning. So we also need a good definition of the whole concept, including for which queue types it applies.

- Support for replications and batched means.

Bibliography

[1] J.F.C.M. de Jongh. *Share Scheduling in Distributed Systems*. Phd thesis, Delft University of Technology, 2002.

[2] Université de Montréal. SSJ: Stochastic Simulation in Java. `http://simul.iro.umontreal.ca/ssj/indexe.html`. Accessed: 2017-09-03.