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 20180429

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 snipplet 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 snipplet 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 `SimEvents` are unique; you cannot schedule a `SimEvent` ↪ more than once on a single `SimEventList`. Typically, `SimEvents` 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 `SimEvents` 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

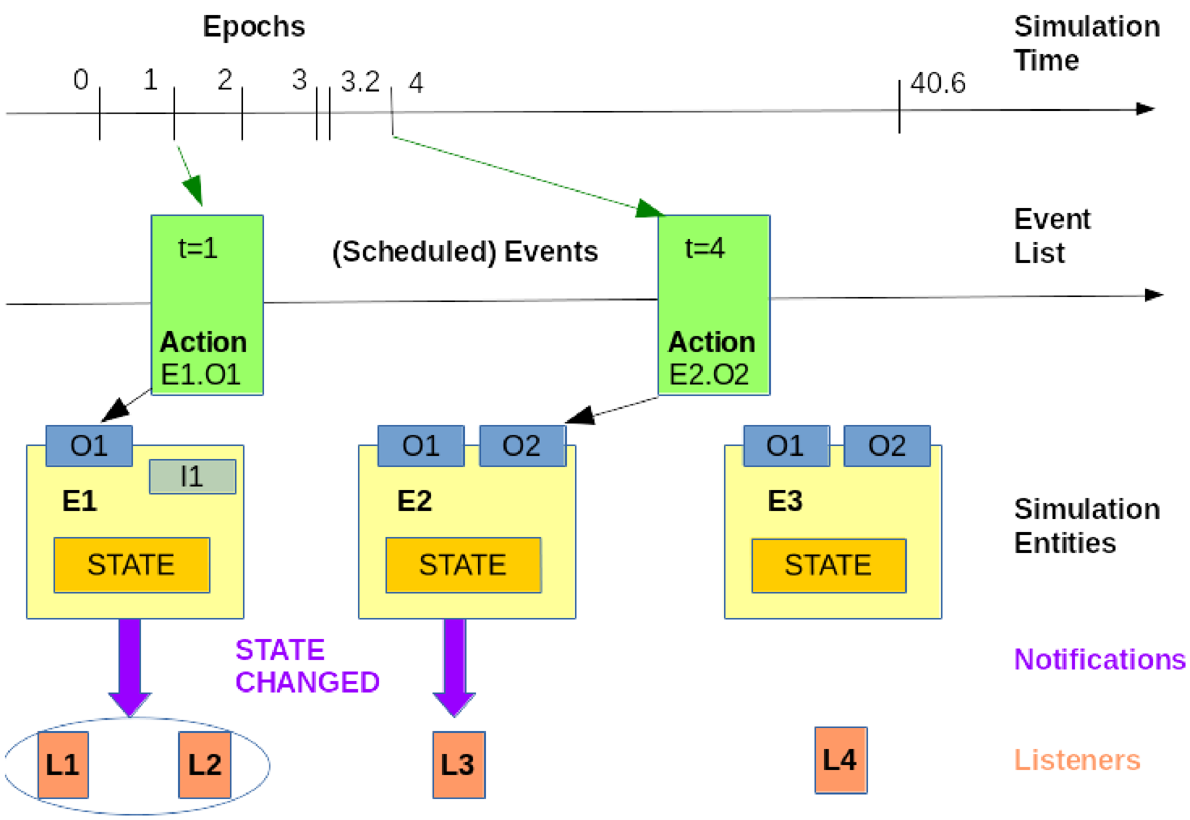
[1]Or until some other criterion is met; see Chapter 4.

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`.



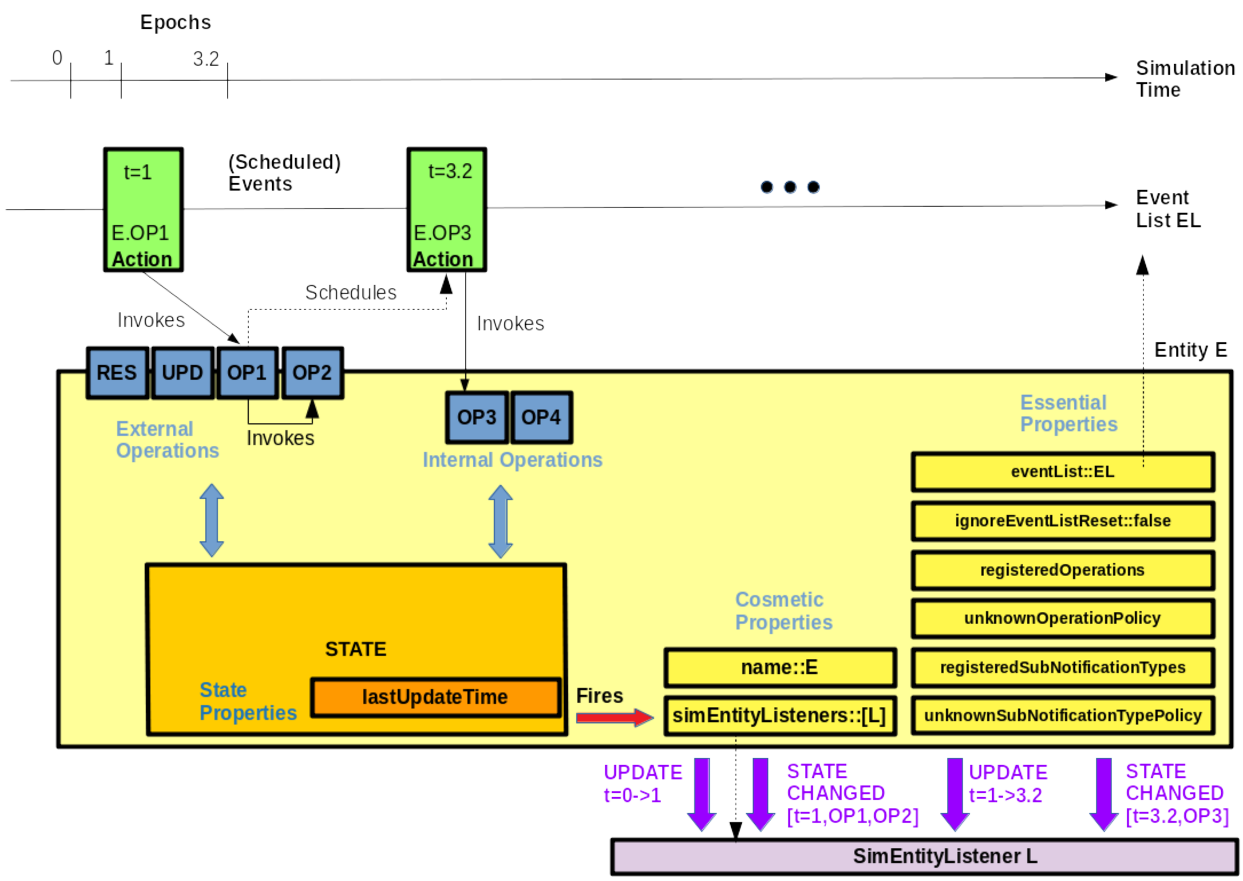
In Figure 5.1, we attempt to explain the `jqueues` concepts and features described thus far. In the top two rows we show the by now hopefully familiar concepts of *epochs*, *simulation time*, *events*, and *actions*. The event list contains two events, one scheduled at simulation time $t = 1$ and one 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.01` invocation at $t = 1$, but notification holds both `E.01` and `E.02` as sub-notifications, because `E.02` was invoked from within the context of `E.01`. 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`.

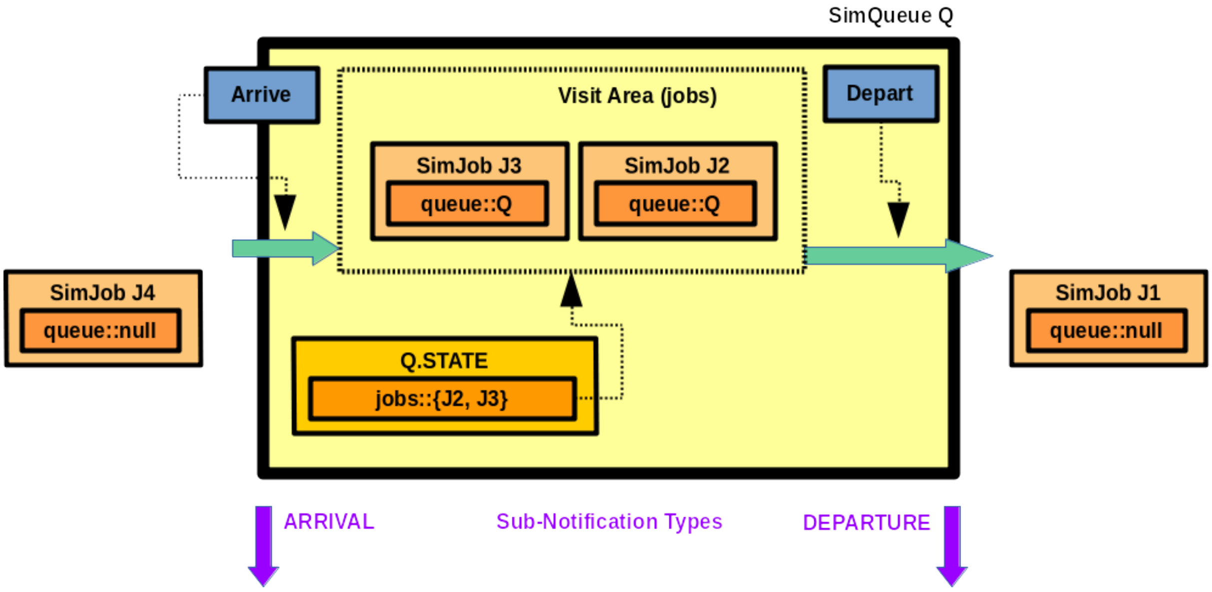
If the visit ends "normally", i.e., the visiting `SimJob` has obtained its required service or some other requirement is met, the `SimQueue` invokes its internal `Depart` operation, with corresponding sub-notification type `DEPARTURE`. Both the `ARRIVAL` and `DEPARTURE` sub-notifications provide the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

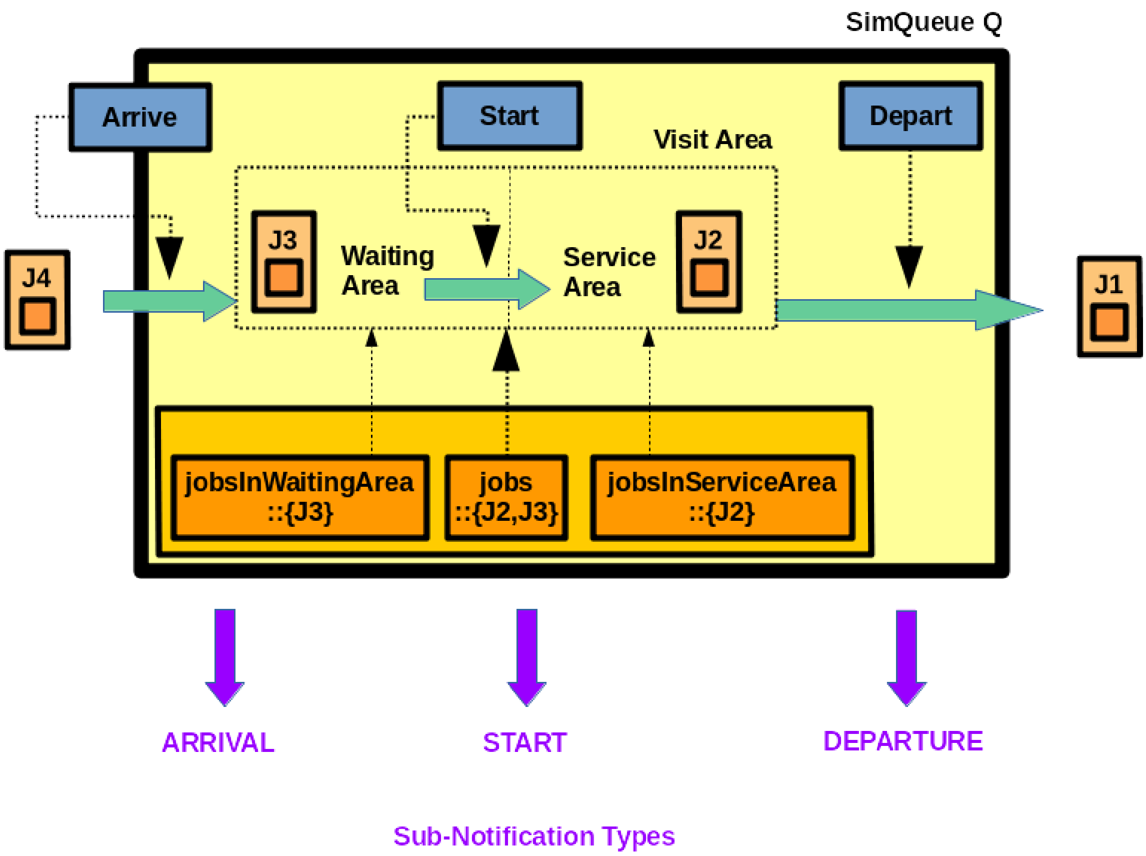
Unlike the case with arrivals, a departure is *not* the only means by which a visit can end. In sections further on, we introduce *dropping*, *revoking* and *auto-revoking* a job as alternative means in which a queue visit can end. We then briefly describe the case in which a visit *does not end at all*, 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 6.7.

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.

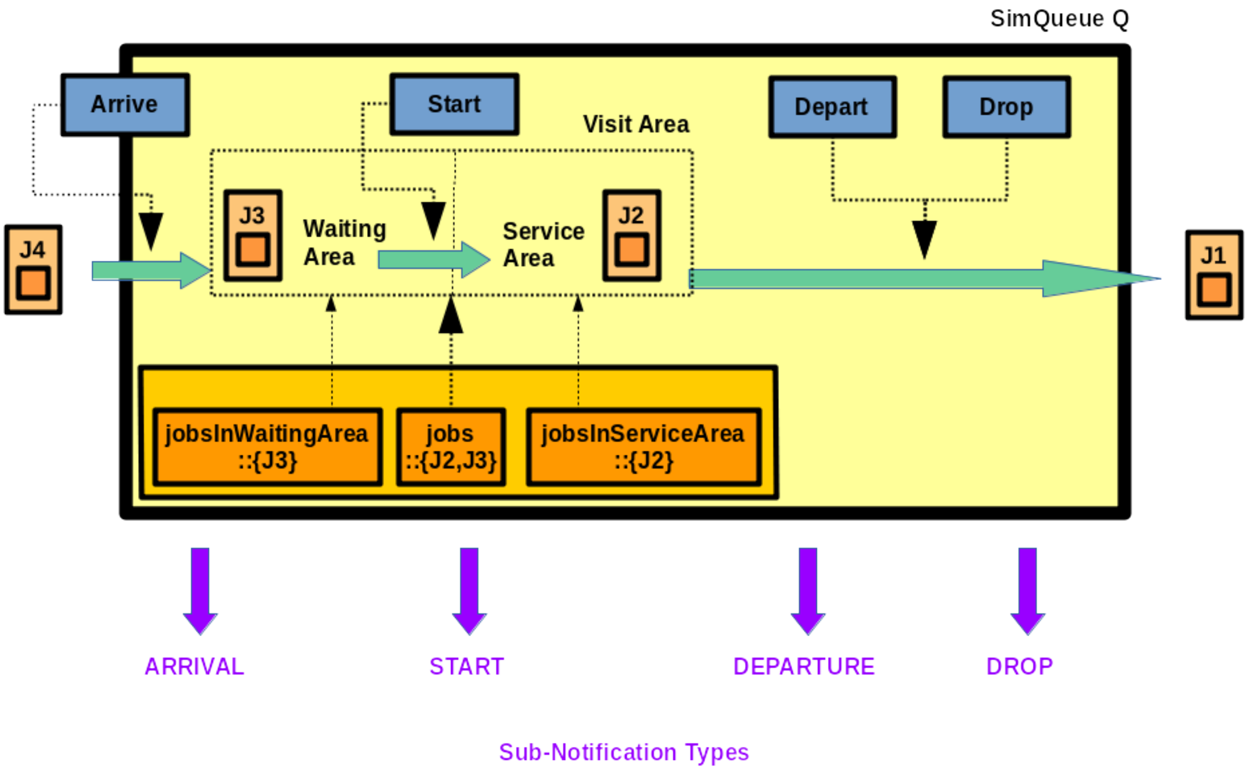
In the modified version of the figure, job `J1` may have *departed* or have been *dropped* from `Q`; the effect of either is the same on `J1`, i.e., a job has no notion of having departed or having been dropped from its latest visited `SimQueue`.

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.

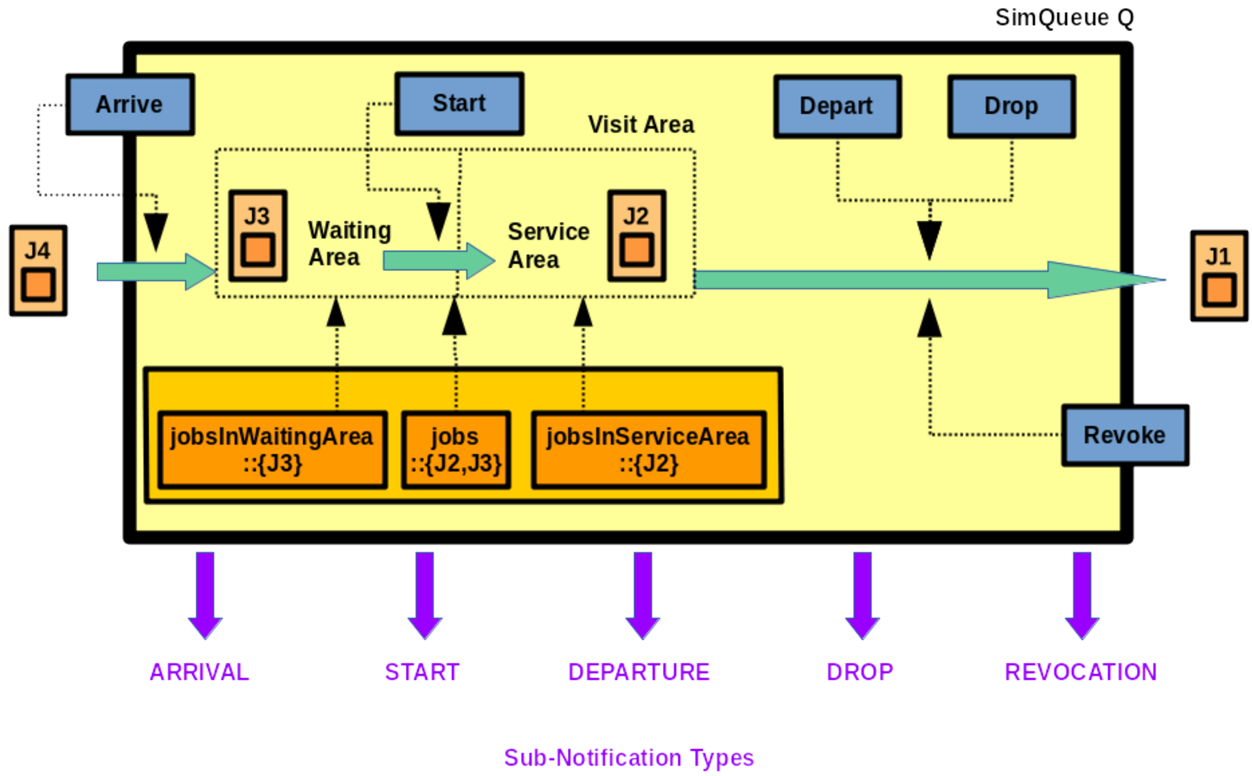
Revocations actually come in two flavors:

- 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.

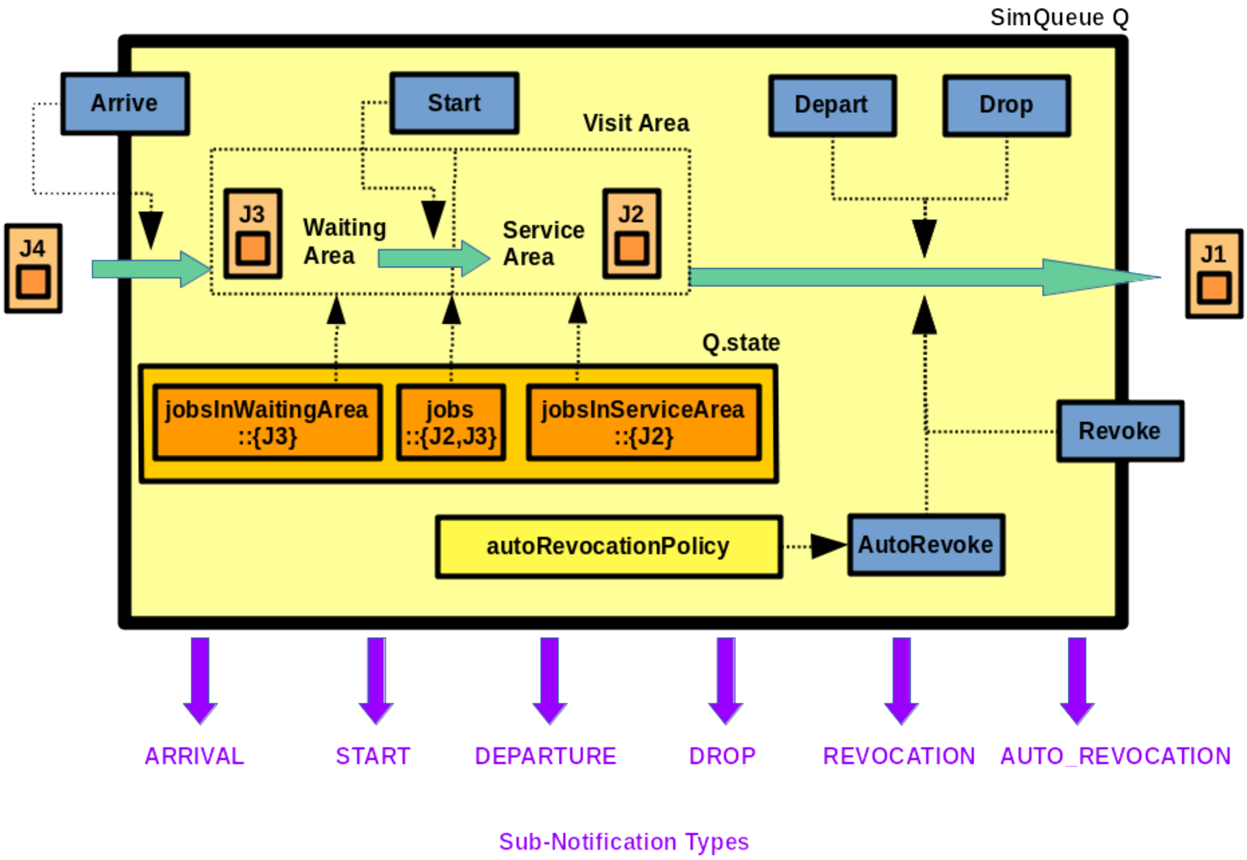
6.5 The AutoRevoke Operation

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.

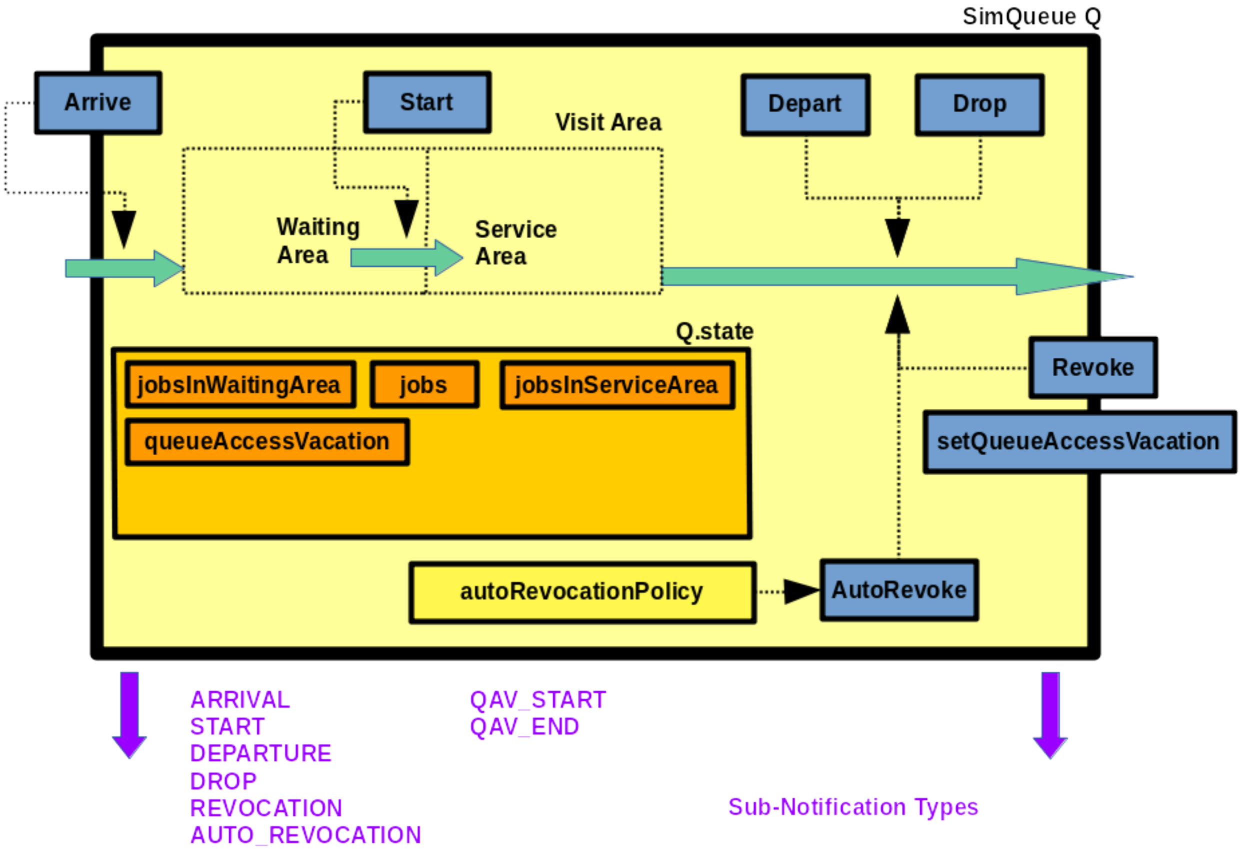
At this point, we have described *all* binary operations, internal as well as external, on `SimJob` and `SimQueue`. In the next sections, we will describe operations, properties and sub-notification types specific to `SimQueue`.

6.6 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 6.6, we show the modified simple model of the state of a `SimQueues` now including Queue-Access Vacations.

Figure 6.6: `SimQueue` model with 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.

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.

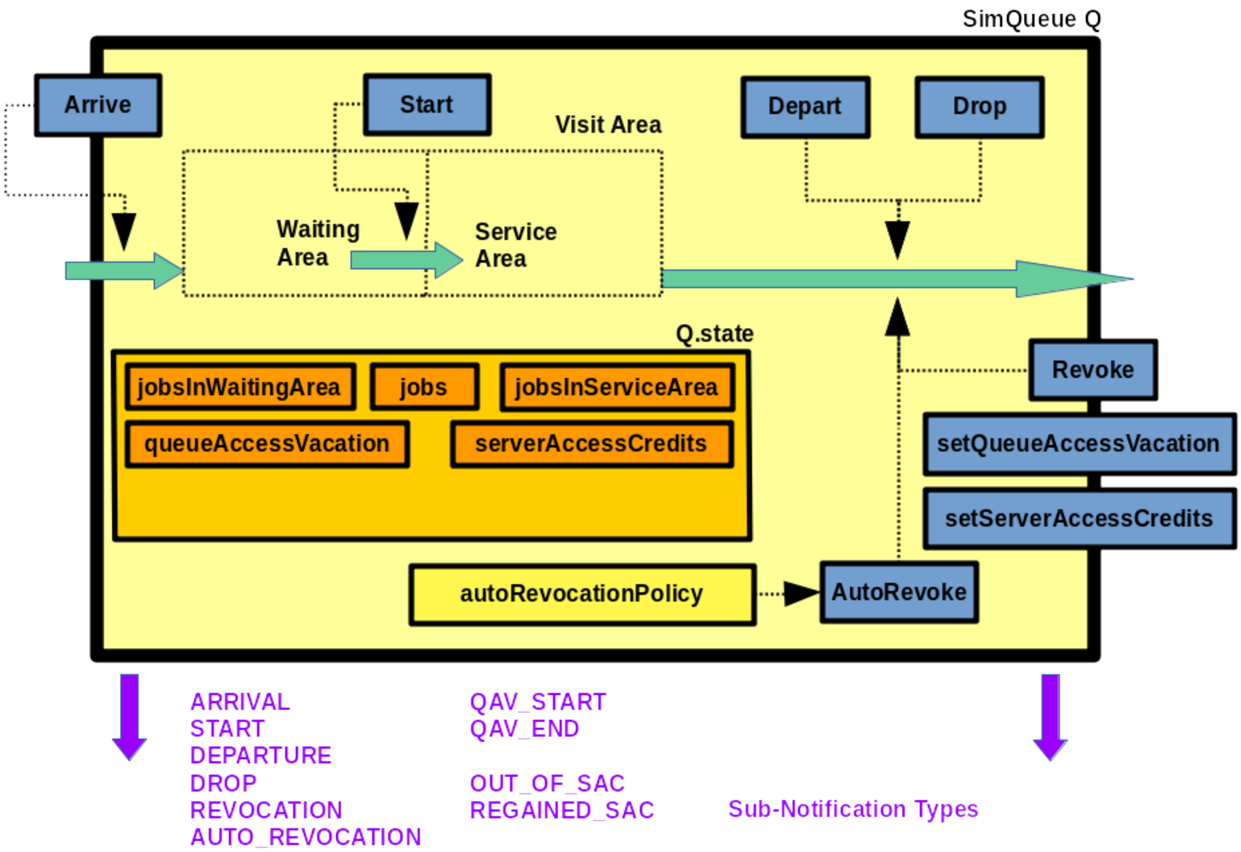
6.7 Server-Access Credits

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 6.7, we show the modified simple model of the state of a SimQueues now including Server-Access Credits.

Figure 6.7: SimQueue model with 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 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.

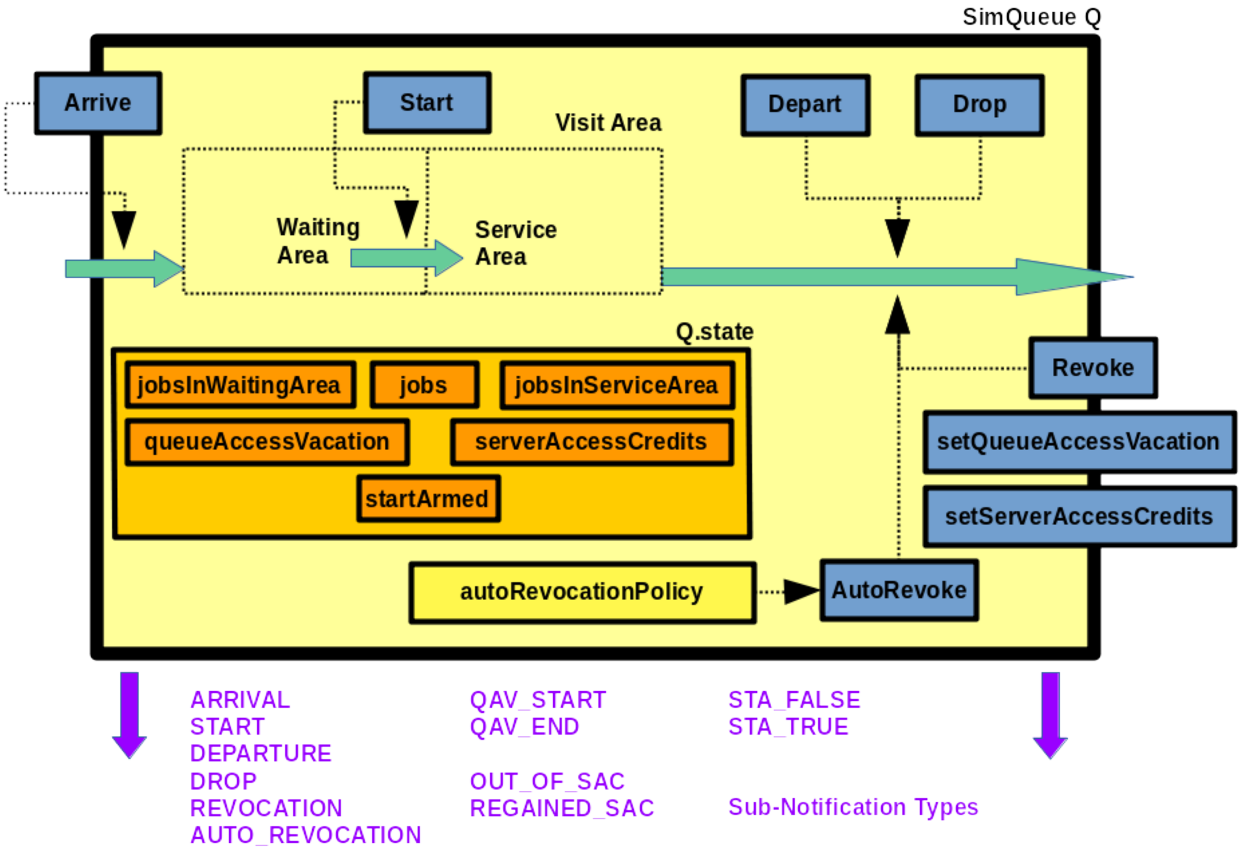
In addition to the default value being ∞, SimQueue implementations cannot use ServerAccessCredits to meet their requirements. For instance, in order to implement queueing systems with multiple servers like FCFS_c (see Section ??), the use of ServerAccessCredits could be queue handy. However, decreasing 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.

6.8 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 6.8, we show the modified simple model of the state of a `SimQueues` now including the `startArmed` state property.

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



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 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.

6.9 The Required 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 requests the job for its *required service time*. To that purpose, each `SimJob` has the `requiredServiceTime` essential property, which has type `Function<SimQueue, Double>`. Its value returns a *required 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.

6.10 Summary of Properties

In Tables 6.1 and 6.2 we list the mandatory properties of a `SimQueue` and a `SimJob`, respectively.

Table 6.1: 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`.
	Cosmetic Properties	
None.		

Table 6.2: Mandatory properties of a SimJob.

Name	Type	Default/Reset
	SimJob	
	Super	
SimEntity		See Table 5.1.
	State Properties	
queue	SimQueue	null.
	Essential Properties	
requiredServiceTime	Function<SimQueue, double>	q->1.0;
	Cosmetic Properties	
None.		

6.11 Summary of Operations

In Table 6.11, we summarize the operations supported on a SimQueue, subdivided into SimEntity ↪ and SimQueue operations.

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.

6.12 Summary of Sub-Notification Types

In Table 6.12, 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.

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.

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

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

Recall that a state change is reported atomically as a `List` of sub-notifications.

Bibliography