

Solution of the simulation design exercise

Discrete event simulation

Below we present the object model of a simulation framework. As the concept of abstract classes forms the basis of every framework we discuss this aspect first.

S.1 Abstract classes and abstract coupling

Classes that define common behavior usually do not represent instantiable classes but abstractions of them. They are called *abstract classes*. It does not make sense to generate instances of abstract classes since some methods are abstract and have empty/dummy implementations. The general idea behind abstract classes is clear and straightforward:

- Properties (that is, instance variables and methods) of similar classes are defined in a common superclass.
- Some methods of the resulting abstract class can be implemented, while only dummy or preliminary implementations can be provided for others, which are termed abstract methods. Though abstract methods cannot be implemented, their names and parameters are specified since descendants cannot change the method interface. So an abstract class creates a *standard class interface* for all descendants. Instances of all descendants of an abstract class will understand at least all messages that are defined in the abstract class.

Sometimes the term *protocol* is used for this standardization property: instances of descendants of a class A support the same protocol as supported by instances of A.

The implication of abstract classes is that other software components based on them can be implemented. These components rely on the protocol supported by the abstract classes, that is they are abstractly coupled with the abstract classes. Most important, these components interact properly, that is without change and recompilation, with instances of all future extensions of the abstract classes.

In the implementation of these components, reference variables that have the static type of the abstract classes they rely on are used. Nevertheless, such

components work with instances of descendants of the abstract classes by means of polymorphism. (A descendant class inherits from another class, but not necessarily as direct subclass.) Due to dynamic binding, such instances can bring in their own specific behavior.

The key problem is to find useful abstractions so that software components can be implemented without knowing the specific details of concrete objects. The case study of a discrete event simulation framework illustrates among other aspects the central role of abstract classes in connection with abstract coupling in the realm of framework design. The simulation framework presented by Reiser and Wirth (1992) inspired this case study and its underlying design.

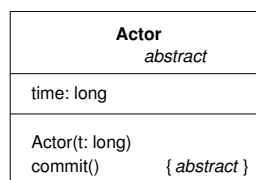
Case study: discrete event simulation

Discrete event simulation is intuitive to understand and can be applied to a variety of problems. Though many think first of modeling waiting lines in stores or banks, discrete event simulation can also form the backbone of network configuration tools and work flow systems, to name just a few.

Events with an associated time represent a central entity in discrete event simulation. Events, such as customer arrivals, are scheduled by a random number generator. The simulation progresses from event to event on the time axis and gathers relevant statistics.

As a consequence, events are one of the key abstractions in a simulation framework. In their presentation of a simulation framework, Reiser and Wirth (1992) call this key abstraction *actor*: “The main simulation program always pairs an action with an event. ... We call that object an actor.” So actors know what to do when they receive certain messages.

Figure S.1 suggests a definition of the abstract class Actor in the Unified Booch & Rumbaugh Notation (Booch and Rumbaugh, 1996). The Additional Reading, Part I summarizes those aspects of the Unified Notation that are relevant in the realm of this white paper. Actors have an instance variable *time* which denotes the due time. The abstract method *commit()* is the only method defined in its protocol. The simulation framework calls this actor method upon due time.



A queue represents a further important simulation entity. For example, customers queue up in front of server stations. As the simulation framework deals with actors, queues handle objects of this type.

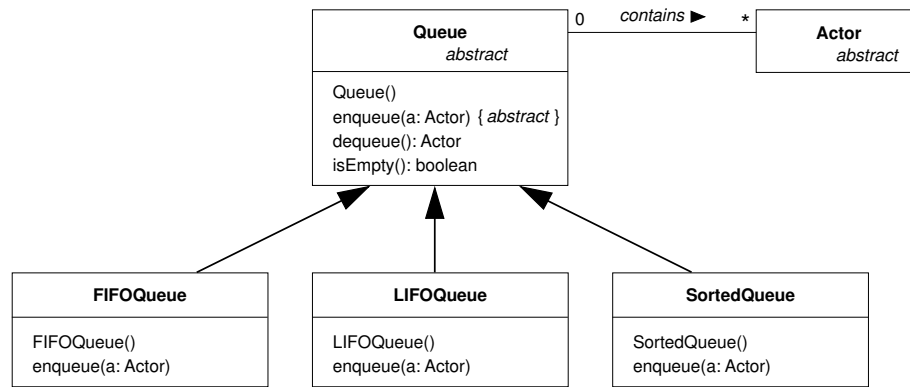


Figure S.2 Queue classes.

A queue can offer the First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) policy for managing queued elements. Based on the time attribute of actors another queue type `SortedQueue` can sort entries according to the due time. These queues differ in the method that enqueues an actor. Figure S.2 shows the interface of the abstract `Queue` class and its specific descendants `FIFOQueue`, `LIFOQueue` and `SortedQueue`.

Besides factoring out common behavior of specific queue types, the abstract class `Queue` is *abstractly coupled* with the abstract class `Actor`. This means that queues work with instances of all future subclasses of `Actor`. Such a simple case of abstract coupling is mainly based on polymorphism as the methods of queues work with any type that is compatible to `Actor`.

Class `Simulation` implements an abstract simulation based on the protocol of `Actor`. Thus `Simulation` is also abstractly coupled with `Actor` (see Figure S.3). The basic idea of the simulation framework is that any discrete simulation system processes events (actors) that are due at a certain point in time. So the core of the simulation framework embodied in the `simulate()` method simply iterates over the actors sorted by the due time, adjusts the current simulation time according to the actor's time and sends the `commit` message to the corresponding actor. The iteration loop ends when the simulation time, that is, the duration of the overall simulation, expires.

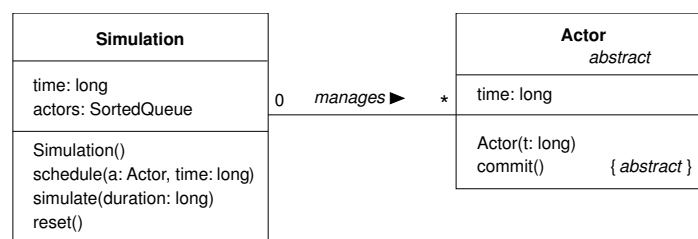


Figure S.3 Abstractly coupled simulation classes.

Thus, a `Simulation` instance stores `Actor` objects in a `SortedQueue` object which is referenced by the instance variable `actors`. The `schedule()` method inserts an actor according to its time into this sorted

queue. Sending the reset message to a Simulation object empties the actor queue and resets the simulation time, that is, assigns 0 to the instance variable time.

Example S.1 shows the Java (Sun, 1996) implementation of method `simulate()` in class `Simulation`.

```

public class Simulation {
    ...
    public void simulate(long duration) {
        Actor actor;
        long endOfSimulation= time + duration;
        do {
            if (!actors.isEmpty()) {
                actor= actors.dequeue();
                time= actor.time;
                actor.commit();
            } else // no more actors enqueued
                time= endOfSimulation+1; // exit loop
        } while (time <= endOfSimulation);
    }
    ...
}

```

To sum up, **Example S.1** Generic simulation loop. the purpose of abstract classes is that other software components based on them can be implemented in advance. Class `Simulation` implements a generic simulation loop based on the protocol of `Actor`. `Simulation` has the important characteristic that an instance of this class interacts properly, that is, without change and recompilation, with any instances of future subclasses of `Actor` that bring in their specific behavior by overriding the `commit()` method.

S.2 White-box versus black-box design

The characteristic of white-box frameworks is that programmers apply inheritance to override methods in subclasses of framework classes in order to accomplish adaptations. On the other hand, black-box frameworks offer ready-made classes whose instances allow adaptations by mere composition. White-box frameworks typically evolve into black-box frameworks over time, that is, after numerous cases of reuse. Below the simulation framework serves as an example of how such an evolution takes place.

The simulation framework so far represents a pure white-box framework in its early design stage. In order to simulate a simple cash desk in a convenience store with a single waiting line, extensions of the basic simulation framework become necessary. These extensions should be designed so that future single-line simulations can reuse them as black boxes.

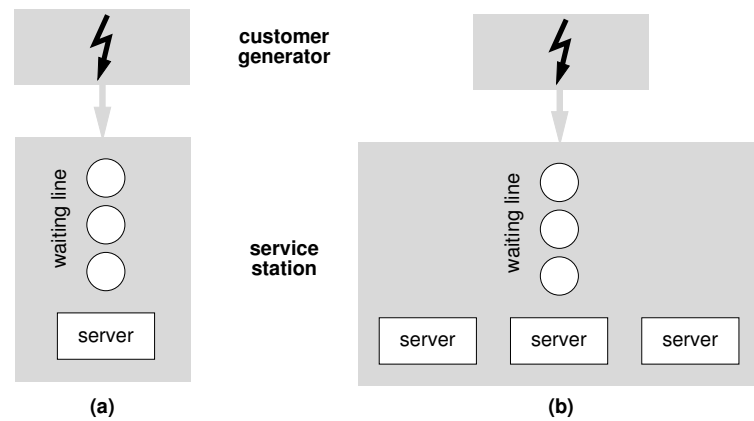


Figure S.4 Single-line simulation with (a) one and (b) several servers.

Figure S.4 illustrates the principal components of a simulation with waiting lines. *Customers* are created randomly by a *customer generator*. A *server* processes customer requests. If the server is busy or not available, customers queue in front of the server. Both the *waiting line* and the corresponding server(s) form one unit. We refer to it as *service station*.

Black-box components on top of a white-box framework

Service stations may differ in several ways. A service station can either offer one or several servers (see Figure S.4 (a) and (b)). Furthermore, in many simulations it is fine to assume that service stations have unlimited space for customers queueing up in the waiting line. In other cases, limitations of the waiting line are necessary. Service stations can, for example, also differ in the queueing policy. Below we discuss the simplest type of service station that has no space limitations, that offers only one server and that queues the customers according to the FIFO principle. The more simulations will be built with the framework the clearer it will be which black-box service stations satisfy simulation needs best. The framework will then comprise numerous service stations so that the chances are high that new simulations just require the appropriate composition of ready-made service stations and customer components. Note that both black-box components, that is, service stations together with customers, rely on the time-dependent notification mechanism provided by the basic white-box framework.

Let us now focus on the *dynamic aspects* of the simulation framework extension. According to Figure S.4, a customer generator creates customers that interact with a service station. Thus a newly created customer requests a service from a service station. If a server is available, the customer is immediately served, otherwise the customer enters the queue. By convention, the message *activate* is sent to the customer whose turn it is as soon as a server is available. The customer, that is, his or her request, determines the service duration. After being served the customer receives the *commit* message upon which he or she frees the server and leaves the simulation system. Figure S.5 illustrates the principal

aspects of the interactions between a customer and its service station by means of an interaction diagram.

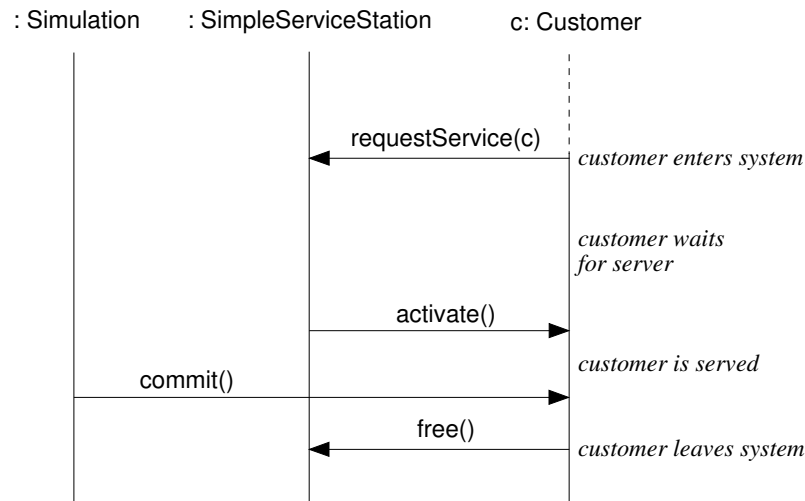


Figure S.5 Interaction between simulation components. S.6 shows the class diagram of SimpleServiceStation and Customer. The reason why Customer is a subclass of Actor, whereas SimpleServiceStation is not derived from Actor is explained below.

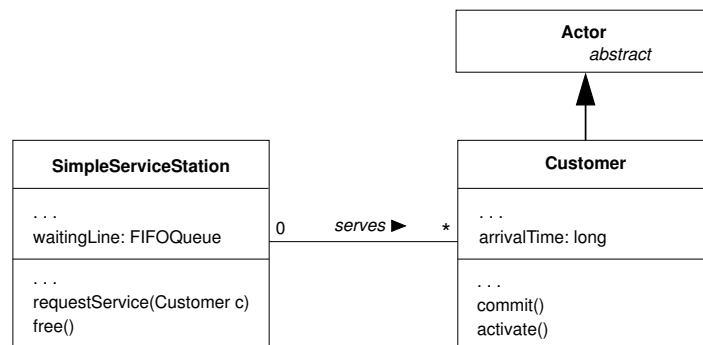


Figure S.6 Classes SimpleServiceStation and Customer. S.2 lists the implementation of the methods requestService(...) and free() of class SimpleServiceStation.

```

public class SimpleServiceStation {
    ...
    public void requestService(Customer c) {
        if (no customer is in the waiting line)
            c.activate(); // server processes customer request
        else
            waitingLine.enqueue(c);
        // gather waiting line statistics
        ...
    }
}
  
```

```

        public void free() {
            Actor actor;
            // gather waiting line statistics
            ...
            if (customers are in the waiting line) {
                actor= waitingLine.dequeue();
                if (actor instanceof Customer)
                    ((Customer)actor).activate();
            }
        }
        ...
    }

```

Example S.2 Implementation details of a simple service station.

Remember class Actor of the basic simulation framework. The idea is that all simulation components which have to be notified at a certain point of time become subclasses of Actor. As the ordinary station discussed above has no time-dependent behavior, such as out-of-service periods, there is no need to define it as subclass of Actor. On the other hand, the customer generator and customers require time-dependent notification.

```

public class CustomerGenerator extends Actor {
    SimpleServiceStation station;
    Simulation simulation;

    public CustomerGenerator(long t, SimpleServiceStation s, Simulation sim) {
        super(t);
        station= s;
        simulation= sim;
    }
    public void commit() {
        Customer c= new Customer(...);
        ...
        station.requestService(c);
        simulation.schedule(new CustomerGenerator(...), // actor
            simulation.time +
            (long)(RandomNumbers.negExp(station.arrivalRate))
        );
    }
    ...
}

```

Example S.3 Implementation details of the customer generator.

A customer generator randomly creates customers. Arrival events follow a negative exponential distribution, parameterized by the average rate of events per time unit. Thus we adapt the simulation framework in method commit() of class CustomerGenerator so that a new customer is created and the next arrival event, that is the next CustomerGenerator actor, is generated and scheduled. Example S.3 lists the relevant aspects of the implementation of class CustomerGenerator. Method negExp(...) is imported from a package RandomNumbers and calculates the

appropriate random number based on the average arrival rate of customers. In order to initialize the simulation, a first customer arrival has to be scheduled by passing a CustomerGenerator object to a Simulation object (see later on in Example S.5).

Customers use the time-dependent notification mechanism of Simulation in the following way: As soon as a customer is activated by a server, the customer schedules its end of service. Thus, the Simulation object invokes commit() when the service period ends, and the customer frees the server in the service station (see Example S.4).

```
public class Customer extends Actor {
    SimpleServiceStation station;
    Simulation simulation;
    long arrivalTime;
    ...
    public Customer(long t, SimpleServiceStation s,
                    Simulation sim) {
        super(t);
        station= s;
        simulation= sim;
        arrivalTime= simulation.time; // for statistical purposes
    }
    public void commit() {
        station.free();
        // gather statistics regarding the waiting time
        ...
    }
    public void activate() {
        simulation.schedule(this,
            (long)(RandomNumbers.negExp(station.avrgServiceDuration));
    }
    ...
}
```

Figure S.7 Implementation details of customers. S.7 shows the class hierarchy of the basic white-box framework and its black-box extensions. Note that the black-box subframework has no abstract classes.

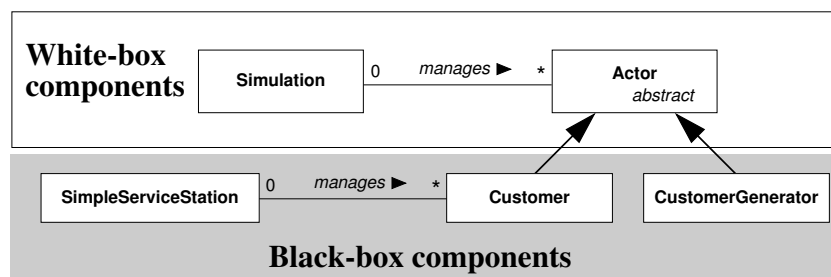


Figure S.7 Black-box and white-box components in the simulation framework.

Black-box composition

The black-box framework components `CustomerGenerator`, `Customer` and `SimpleServiceStation` together with the generic time-dependent notification mechanism of the basic framework suffice to compose numerous simulations that are based on the specific type of service station supported by class `SimpleServiceStation`. The source code of method `simulate(...)` in class `ConvenienceStoreSim` (see Example S.5) does such a sample composition.

```
public class ConvenienceStoreSim {
    ...
    public void simulate(int avrgServiceDuration, int arrivalRate, int duration) {
        SimpleServiceStation station;
        Simulation simulation;
        CustomerGenerator generator;

        simulation= new Simulation();
        station= new SimpleServiceStation(simulation, arrivalRate,
                                           avrgServiceDuration, ...);
        generator= new CustomerGenerator(0, station, simulation;
        simulation.schedule(generator,
                               0); // start with the customer generation
        simulation.simulate(duration);

        station.provideStatistics();
    }
    ...
}
```

Example S.5 Composition of a point-of-sale simulation of a convenience store.

Such a configuration lends itself to be accomplished by end users without having to write one line of code. End users could specify simulation scenarios by interactive GUI editors. These editors would offer all available black-box components for compositions.

Design considerations

The black-box classes in the current development stage of the simulation framework represent components that can be reused very easily if their design and implementation fit exactly. If this is not the case, programmers would develop further black-box classes that extend the abstract classes of the basic simulation framework. What does this imply for a further design refinement of the simulation framework?

First of all, the framework would require several service station types that will be implemented analogous to the simple one discussed above. Probably, these stations have enough in common so that a further abstract class, for example, called `ServiceStation`, factors out the commonalities. If programmers don't find the appropriate station type in the framework, they could develop the adequate one by just specifying the differences to this more general station type.

So not all station type implementations would have to start from scratch. As the development of further station types would go beyond the scope of this white paper, we refrain from coming up with an appropriate abstract class for service stations.

Another problem of the current design is that the classes Customer, CustomerGenerator and SimpleServiceStation are tightly coupled. What does tight coupling mean in this context and what are its implications? SimpleServiceStation instances rely on the protocol of Customer objects and vice versa. A Customer object provides the methods commit() (inherited and overridden from Actor) and activate(). A service station sends the activate message in order to signal a customer that a server is available and the service can start. On the other hand, objects of the classes CustomerGenerator and Customer rely on the protocol of SimpleServiceStation. Now think of a station type which also allows the simulation of server break downs. Probably, customers should be notified of break downs. Thus, not only the station type but also a specific customer and customer generator class have to be developed. In other words, future simulation requirements that cannot be handled by the available black-box classes might always imply the development of all three simulation entities.

Further case studies in the realm of the workshop discuss means to make frameworks more flexible. Some of the proposed modifications of the simulation framework partially loose the rigidity of the black-box classes and thus contribute to the decoupling of these simulation entities.

S.3 References

Booch G. and Rumbaugh J. (1996) *Unified Method*. Documentation Set, Santa Clara, CA: Rational Software Corporation.

Reiser M. and Wirth N. (1992). *Programming in Oberon—Steps Beyond Pascal and Modula*. Wokingham: Addison-Wesley/ACM Press

Sun microsystems (1996). Java's WWW home page at <http://java.sun.com>