27

# 27 Example: Supermarket

This chapter presents a single example program. The program is a simulation, slightly more elaborate than that in the AirController/Aircraft example in Chapter 20. The program makes limited use of inheritance. The use of inheritance here is largely to simplify some aspects of the design; there isn't much sharing of code among interrelated classes.

The number of objects present at run time is somewhat larger in this example than earlier examples. At any one stage of the simulation there may be as many as two hundred to three hundred objects. The number of objects created and destroyed during a run will be something in the range one thousand to three thousand. Naturally, since the program will be creating and working with large numbers of objects, some of the standard collection classes appear. Although there are hundreds of objects most are instances of the same class; the program uses fewer than ten different concrete classes.

The program makes use of a class that came with the very first class library released with C++. This "histogram" class (based on the version in the original "tasks" library) collects data that are to be displayed as a histogram and, when data collection is completed, produces a printout.

## 27.1 BACKGROUND AND PROGRAM SPECIFICATION

### Helping the supermarket's manager

The manager of a large supermarket wants to explore possible policies regarding the number of checkout lanes that are necessary to provide a satisfactory service to customers.

The number of customers, and the volume of their purchases, varies quite markedly at different times of day and the number of open checkout lines should be adjusted to match the demand. Customers will switch to rival stores if they find that they have to queue too long in this supermarket so, at times, it may be necessary to have a large number of checkouts open. However, the manager must also try to minimize idle time at checkouts; there is no point keeping checkouts open if there are no customers to serve.

The manager is proposing a policy in which supermarket staff will be assigned to "general duties". Staff may be restocking shelves, cleaning, recovering trolleys from parking lots, or operating checkouts. Staff can be switched "immediately" between different roles. The manager (or some deputy) will make regular checks on activity in the shop and, if appropriate, may assign more staff to open extra checkout lanes or may close idle checkouts so freeing the operators to perform other duties.

***Rules for running the supermarket***

The manager is proposing some "rules of thumb" (heuristics) for choosing when to open or close checkouts and for the frequency of scheduling such decisions. However, before trying these rules in the shop, the manager wishes to try them out in simulations.

***Rule parameters***

The rules are "parameterized". For example, one rule limits use of "fast checkout lanes" to customers who are purchasing less than some specified maximum number of items. Naturally, this "fast checkout limit" is a parameter that can be adjusted. The manager wants to run many simulations with different settings for the various parameters. In this way, it is hoped that it will be possible to identify which parameters are more critical and to establish some practical operating rules.

***Program to simulate working for different sets of parameters***

The program, that is to be developed for the manager, is to start by accepting a set of inputs that define a particular set of control parameters for the manager's rules. It is then to simulate one day of operation of the supermarket subject to these rules. The simulation will involve some "randomized" customer traffic. During the simulation, various statistics will be gathered that characterize the distribution of queuing times for the customers and the total amount of open-time and idle time for the checkouts. At the end of a simulation run, these data are to be displayed. The manager may then wish to initiate another "randomized" run, or may wish to repeat the run using the same "randomized" pattern for customer traffic but with a changed set of parameters for the rules.

<u>Some details:</u>

***Time period simulated***

The supermarket opens its doors at 8 a.m.. Once opened, the doors can be pictured as delivering another batch of frantic shoppers every minute. This continues until the manager closes the doors, which will happen at the first check time after the supermarket's nominal closing time of 7 p.m.. All customers then in the supermarket are allowed to complete their shopping, and must queue to pay for their purchases before leaving. During this closing period, the manager checks the state of the shop every 5 minutes and closes any checkouts that have become idle. The supermarket finally closes when all customers have been served. The closing time should be displayed along with the statistics that have been gathered to characterize the simulation run.

***Checkout numbers***

The supermarket has a maximum of 40 checkout lanes, but the maximum number to be used in any run of the simulation is one of the parameters that the manager wishes to vary. The checkouts may be "fast" or "standard" (as noted earlier, another of the parameters for a simulation run defines the limit on purchases permitted to users of fast checkout lanes). The supermarket always has at least one standard checkout open; the manager wants the option of specifying a minimum of 1..3. There is no requirement that a "fast" checkout be always open, though again the manager wants to be able to specify a minimum number of fast checkouts (in the range 0..3). One of the other rules proposed by the manager specifies when to open fast checkouts if there are none already open.

***Queues at checkouts***

Checkouts process a current customer, and have a "first-come-first-served" queue of customers attached. A checkout processes ten items per minute (this is not one of the parameters that the manager normally wishes to vary, but obviously the processing rate, as defined in the program, should be easy to change.) The minimum processing time at a checkout is one minute (customers have to find their change and dispute the bill even if they buy only one item). Once a checkout finishes processing a customer, that customer leaves the shop (and disappears from the simulation). Checkout operators record any time period that they are idle and, when reassigned to other duties, report their idle time to a supervisor who accumulates the total idle time.

***Customers***

Customers entering the supermarket have some planned number of purchases (see below for how this is determined). Customers have to wander the aisles finding the items that they require before they can join a queue at a checkout. It is a big store and the minimum time between entry and joining a checkout queue would be two minutes for a customer who doesn't purchase anything. Most customers spend a reasonable amount of time doing their shopping before they get to join a queue at a checkout. This shopping time is determined by the number of items that must be purchased and the customer's "shopping rate". Shopping rates vary; for these simulations the shopping rates should be distributed in the range 1..5 items per minute (again, this range is not a parameter that the manager wishes to change on different runs of the simulation, but the program should define the range in a way that it is easy to change if necessary).

***Choosing where to queue***

Once they have completed their shopping, customers must choose the checkout where they wish to queue. You can divide customers into "fast" and "standard" categories. "Fast" customers are those who are purchasing a number of items less than or equal to the supermarket's "fast checkout limit" (together with that 1% of other customers who don't wish to obey such constraints). When choosing a checkout, a "fast customer" will, in order of preference, pick: a) an idle "fast" checkout, b) an idle "standard" checkout, or c) the checkout ("fast" or "standard") with the minimum current workload. Similarly, "standard customers" will pick a) an idle "standard" checkout, or b) the "standard" checkout with the minimum current workload.

***Workloads at checkouts***

The workload of a checkout can be taken as the sum of all the items in the trolleys of customers already in the queue at that checkout plus the number of items still remaining to be scanned for the current customer.

***Scheduling changes to checkouts***

The manager is proposing a scheme whereby checks are made at regular intervals. This interval is one of the parameters for a simulation run; it should be something in the range 5..60 minutes.

When running a check on the state of the supermarket, the manager may need to close idle checkouts, or to open checkouts to meet the minimum requirements for fast and standard checkouts, or to open extra checkouts to avoid excessively long queues at checkouts. The suggested rules are:

***Rules for opening/closing checkouts***

- Checkout closing rule:
  If the number of shoppers in the aisles has decreased since the last check, then all idle checkouts should be closed (apart from those that need to be left open to meet minimum requirements).

- Minimum checkouts rules:
  Open fast and standard checkouts as needed to make up the specified minimum requirements.
  If the total number of customers in the supermarket (shoppers and queuers) exceeds 20, there must be at least one fast checkout open.

- Extra checkouts rules:

  If either the number of customers queuing, or the number of customers shopping has increased since the last check, then the following rules for opening extra checkouts should be applied:

  a) If the average queue length at fast checkouts exceeds a minimum (specified as an input for the simulation run), then one additional fast checkout should be opened (but not if this would cause the total number of checkouts currently open to exceed the overall limit).

  b) A number of additional standard checkouts may have to be opened (subject to limits on the overall number of checkouts allowed to be open). Checkouts should be opened until the average queuing time at standard checkouts falls below a maximum limit entered as an input parameter for the simulation. (The queuing times can be estimated from the workloads at the checkouts and the known processing rate of checkouts.)

In the real world, when a new checkout opens, customers at the tails of existing queues move to the queue forming at the newly opened checkout if this would result in their being served more quickly. For simplicity, the simulation will omit this detail.

***Customer traffic***

The rate of arrival of customers is far from uniform and cannot be simulated by random drawing from a uniform distribution. Instead, it must be "scripted". The pattern of arrivals averaged for a number of days serves as the basis for this script. Typically, there is fairly brisk traffic just after opening with people purchasing items while on their way to work. After a lull traffic builds to a peak around 10.30 a.m. and then slackens off. There is another brisk period around lunch time, a quiet afternoon, and a final busy period between 5.30 p.m. and 6.30 p.m.. The averaging of several days records has already been done; the results are in a file in the form of a definition of an initialized array `Arrivals[]`:

```
static int Arrivals[] = {
// Arrivals at each 1 minute interval starting 8am
 12,  6,  2,  0,  0,  1,  2,  2,  3,  4,  3,  0,  2,  0,  0,
 ...
};
```

This has entries giving the average number of people entering in each one minute time interval. Thus, in the example, 12 customers entered between 8.00 and 8.01 am etc. These values are to be used to provide "randomized" rates of customer arrival. If for time period i, `Arrivals[i]` is *n*, then make the number of customers entering in that period a random number in the range *0..2n-1*.

***Purchase amounts***

Similarly, the number of purchases made by customers is far from uniform. In fact, the distribution is pretty close to exponential. Most customers buy relatively few items, but a few individuals do end up filling several trolleys with three hundred or more items. The number of items purchased by customers can be approximated by taking a random number from an exponential distribution with a given mean value.

It has been noted that the mean values for these distributions are time dependent. Early customers, and those buying items for their lunch, require relatively few items (<10); so at these times the means for the exponential distributions are low. The customers shopping around 10.30 a.m., and those shopping between 5.30 and 6.30 p.m., are typically purchasing the family groceries for a week and so the mean numbers of items at these times are high (100+).

Again, these patterns are accommodated through scripting. The file with the array `Arrivals[]` has a commensurate array `Purchases[]`:

```
static int Purchases[] = {
// Average number purchases of customer
 4,  3,  4,  5,  3,  5,  7,  6,  5,  4,  3,  5,  2,  5,  4,
…
};
```

The entries in this array are the mean values for the number of purchases made by those customers entering in a particular one minute period. The number of purchases made by an individual customer should be generated as a random number taken from an exponential distribution with the given mean.

***Reports for the boss***

The manager wants the program to provide an informative statistical summary at the end of each run. This summary should include:

- the shop's closing time;
- the number of customers served;
- a histogram showing the number of items purchased by customers;
- histograms summarizing the shopping, queuing and total times spent by customers;
- details of checkout operations such as total operating and idle times of checkouts

There should also be a mechanism for displaying the state of the supermarket at each of the check times.

### Specification

Implement the Supermarket program:

1 The program will start by prompting for the input parameters:
- frequency of floor manager checks;
- maximum number of checkouts;
- minimum number of fast and standard checkouts;
- purchase limit for use of fast checkout;
- length of queues at fast checkout necessary that, if exceeded, will cause the manager to open an extra fast checkout;
- queuing time at standard checkout that, if exceeded, will cause the manager to open an extra fast checkout;

2 The program is then to simulate activities in the shop from 8.00 a.m. until final closing time.
At intervals corresponding to the floor manager's checks, the program is to print a display of the state of the shop. This should include summaries of the total number of customers currently in the shop, those in queues, and details of the checkouts. Active checkouts should indicate their queue lengths. The number of idle checkouts should be stated.
Details of any changes to the number of open checkouts should also be printed.

3 When all customers have left, the final closing time should be printed along with the histograms of the statistics acquired. The histograms should include those showing number of purchases, total time spent in the shop, and queuing times. Details of total and idle time of checkouts should also be printed.

## 27.2 DESIGN

### 27.2.1 Design preliminaries

#### The simulation mechanism

***Loop representing passage of time***

The simulation mechanism is similar but not identical to that used in the AirController/Aircraft example. The core of the simulation is again going to be a loop. Each cycle of this loop will represent one (or more) minute(s) of simulated time. Things happen almost every minute (remember, another bunch of frantic customers comes through the door). Most activities will run for multiple minutes (two minute minimum shopping time, one minute minimum checkout time etc).

***On each cycle do …***

Each cycle of the simulation loop should allow any object that needs to perform some processing to "run". This is again similar to the AirController/Aircraft example where all the `Aircraft` got a chance to `Move()` and update their positions. The Supermarket program could arrange to tell all objects to "run" for one minute (shoppers complete more of their purchasing, checkouts scan a few more items). However, as there are now going to be hundreds of objects it is better to use a slightly more efficient mechanism whereby objects suspend themselves for a period of time and only those that really need to run do so in any one cycle.

***Priority queue used in simulation***

There is a standard approach for simulations that makes use of a priority queue. Objects get put in this queue using "priorities" that represent the time at which they are going to be ready to switch to a new task (all times can be defined as integer values – minutes after opening time). For example, a customer who starts doing 15 minutes worth of shopping at 8.30 a.m. can be inserted into this queue with priority 45 (i.e. ready at 8.45 a.m.), a customer starting at the same time but with only 3 minutes worth of shopping will get entered with priority 33 (i.e. ready at 8.33).

The simulation loop doesn't have to advance time by single units. Instead, it pulls an item from the front of the priority queue. Simulated time can then be advanced to the time at which this next event is supposed to happen. So, if the front item in the queue is supposed to occur at 8.33 a.m., the simulated time is advanced to 8.33 a.m.. In this example, there will tend to be activities scheduled for every minute; but you often have examples where there are quiet times where nothing happens for a period. The priority queue mechanism lets a simulation jump these quiet periods.

When items are taken from the queue, they are given a chance to "run". Now "run" will typically mean finishing one operation and starting another, though for some objects it will just mean doing the same thing another time. This mechanism for running a simulation is efficient because "run" functions are only called when it is time for something important to happen. Thus, there no need to disturb every one of the hundred customers still actively shopping to ask if they are ready go to a checkout; instead, the customers are scheduled to be at the front of the priority queue when they are ready to move to checkouts.

Usually, the result of an object's "run" function will be some indication that the it wants to be put back in the priority queue at some later time (larger priority number), or that it needs to be transferred to some other queue, or that it is finished and can be removed from the simulation.

There will be something in the program that handles this main loop. But before we get too deep into those details we need to identify the objects. What are these things that get to run, and what do they want to do when they have a chance to run?

## The objects

So, what are the objects? What they own? What they do?

***Obvious objects***

Some are obvious. There are going to be "Customer" objects and "Checkout" objects. There will also be "Queue" objects, and "Histogram" objects. Now while all these will be important it should be clear that they aren't the ones that really define the overall working of the simulation.

***Customers***

`Customer` objects are going to be pretty passive. They will just hang around "shopping" until they eventually chose to move to a checkout queue. They will then hang around in a queue until they get "processed" by a `Checkout` after which they will report their shopping time, queuing time, etc and disappear. But something has got to create them. Something has to ask them for their statistics before they leave. Something has to keep count of them so that these data are available when the rules for opening/closing checkouts are used.

***Checkouts***

`Checkout` objects might be a little more active. They will add customers to their queues (if customers were to be permitted to switch queues, the checkouts would have to allow customers to be removed from the tails of their queues). They pull customers off their queues for processing. They report how busy they are. But they still don't organize much. There has to be some other thing that keeps track of the currently active checkouts.

***Finding the objects***

The other objects, in this example the more important control objects, have still to be identified.

***Underline the nouns?***

An approach sometimes suggested is to proceed by underlining the nouns in the problem description and program specification. After all, nouns represent objects. You will get a lengthy and rather strange list: supermarket, door, minute, time, shopping, purchases, manger, employee, statistics, aisles, …. You drop those that you feel confident don't add much; so minute, shopping, aisles can all go immediately. The remaining nouns are considered in more detail.

***Purchases?***

How about purchases? Not an object. The only thing we need to know about a customer's purchases is the number. Purchases can be a data member of the `Customer` class. The class had better provide an access function to let other objects get this number when needed.

***Time?***

Time? The simulation has to represent time. But basically it is just an integer counter (minutes after opening time or maybe minutes after midnight). Something owns the system's timer; something updates it (by advancing this timer when items are taken from the priority queue). Many objects will need access to the time value. But the time is not an object.

***Manager?***

Manager? This seems a better candidate. Something needs to hold the parameters used in the checkout opening/closing rules. Something needs to execute the code embodying those rules. There would only be one instance of this class. Despite that, it seems plausible. It offers a place to group some related data and behaviours.

***Employee?***

Employees? No. The simulation doesn't really need to represent them. Their activities when they are not operating checkouts are irrelevant to the simulation. There is no need to simulate both employee and checkout. In the simulation the checkouts embody all the intelligence needed to perform their work.

***Avoid over faithful models of real world***

You can do design by underlining nouns, and then choosing behaviours for the objects that these nouns represent. It doesn't always work well. Often it results in over faithfully modelling the real world. You tend to end up with "Employee" objects and `Checkout` objects, and a scheme for assigning "Employee" objects to run `Checkout` objects. This just adds unnecessary complication to the program.

***Alternative approach for finding objects: Scenarios***

Use of scenarios is an alternative way to find objects. Scenarios, and the patterns of object interactions that they reveal, were used in Chapter 22. There we already had a good idea as to the classes and needed just to flesh out our understanding of their behaviours and interactions. However, we can use scenarios at an earlier stage where we are still trying to identify the classes. Sometimes, a scenario will have a "something" object that asks a `Customer` object to provide some data. We can try and identify these "somethings" as we go along.

***Starting points for scenarios***

The example programs in Chapter 22 were driven by user entered commands. So, we could proceed by working out "What happens when the user requests action X?" and following the interactions between the objects that were involved in satisfying the users request.

This program isn't command driven. It grabs some initial input, then runs itself. So we can't start by following the effect of each command.

Where else might we start?

The creation and destruction of objects are important events in any program. We have already identified the need for `Customer` objects and `Checkout` objects. So, a possible starting point is looking at how these get created and destroyed.

***Scenarios for the creation and destruction of objects***

Something has to create `Customers`, tell them how much they want to buy, and then pass them to some other something that looks after them until they leave the shop. Note, we have already found the need for two (different kinds of) somethings. The first something knows about those tables of arrivals and purchase amounts. The second something owns data like a list of current customers.

When `Customer` objects leave, they have to report their statistics. How? To what? A `Customer` object has several pieces of data to report – its queuing time, its number of purchases, its shopping time, etc. These data are used to update different `Histogram` objects. It would be very inconvenient if every `Customer` needed to know about each of the different `Histogram` objects (too many pointers). Instead, the gathering of statistics would be better handled by some object that knew about `Customer` objects and also knew about (possibly owned?) the different `Histogram` objects.

`Checkout` objects also get created and destroyed. The "Manager" chooses when creation and destruction occurs and may do the operations or may ask some other object to do the actual creation (destruction). Something has to keep track of the checkout objects. When a checkout is destroyed, it has to report its idle time to something.

Scenarios relating to these creation and destruction steps of known objects will help identify other objects that are needed. Subsequently, we can examine scenarios related to the main simulation loop. "Active objects" are going to get taken from the front of the priority queue and told to "run". We will have to see what happens when objects of different types "run".

### 27.2.2 Scenarios: identifying objects, their classes, their responsibilities, their data

#### Creation and destruction scenarios

<u>Creating customers</u>

A supermarket without customers is uninteresting, so we might as well start by looking at how Customer objects come into the system.

Figure 27.1 provides a first idea. We could have a "Door" object that creates the `Customers`, providing them with the information they need (like their number of purchases). The `Customer` objects will have to complete any initialization (e.g. choose a rate at which to shop), work out how long their shopping will take, and then they will have to add themselves to some collection maintained by a "Shop" object. To do that, they will have to be given a pointer to the shop object (customers need to know which shop they are in). The `Door` object can provide this pointer (so long as it knows which shop it is attached to).

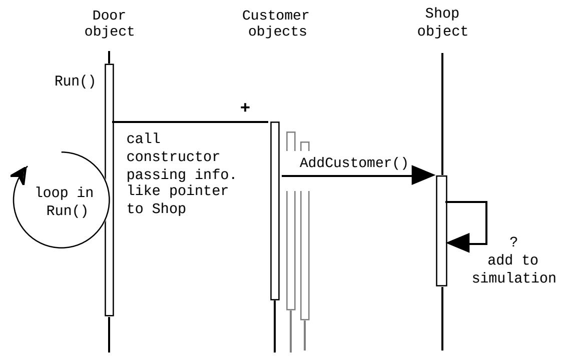


Figure 27.1 Scenario for creation of Customer objects.

Customers are supposed to come into the shop every minute. The main simulation loop is going to get "active objects" (like the `Door`) to "run" each minute (simulated time). Consequently, the activities shown in Figure 27.1 represent the "run" behaviour of class `Door`. It can use the table of arrival rates and average purchases to choose how many `Customer` objects to create on each call to its `Run()` function.

This initial rough scenario for creating a `Customer` object cannot clarify exactly what the `Shop` object must do when it "adds a customer". Obviously it could have a list of some form that holds all customers; but it might keep separate lists of those shopping and those queuing. Some activity by the customer (switch from shopping to queuing) has also got to be scheduled into the simulation. The `Shop` object might be able to organize getting the customer into the simulation's priority queue.

Results:

We appear to need:

- a `Door` object.
  It uses (owns?) those arrays (with details of when customers arrive and how much they want to purchase), and a pointer to a `Shop` object.
  In its `Run()` member function, called from the main simulation loop, it creates customers.

- Constructor
  The constructor function gets given a pointer to a `Shop`, and a value for the number of purchases. The function is to pick an actual number of purchases (exponentially distributed with given number as mean, minimum of 1 item), and select a shopping rate. The `Customer` better record starting time so that later it is possible to calculate things like total time spent in the shop. (How does it get the current time? Unknown, maybe the time is a global, or maybe

the Customer object could ask the Shop. Decide later.) These data get stored in private data members.

Constructor should invoke "AddCustomer" member function of the Shop object.

- a Shop object.
This keeps track of all customers, and gets them involved in the main simulation.

Removing customers from simulation

Customers leave the simulation when they finish being processed by a Checkout. The Checkout object would be executing its Run() member function when it gets to finish with a current customer; it can probably just delete the Customer object. We can arrange that the destructor function for class Customer notify the Shop object. The Shop object can update its count of customers present. Somehow, the Customer has to report the total time it spent in the shop. Probably the report should be made to the Shop object; it can log these times and use the data to update the Histogram objects that will be used to display the data.

Figure 27.2 illustrates the interactions that appear to be needed.

Results:

Responsibilities of Shop object becoming clearer. It is going to gather the statistics needed by the various Histogram s. It probably owns these Histogram objects.

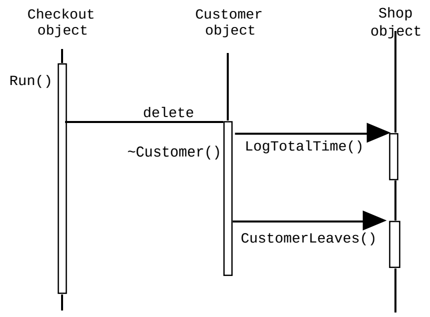


Figure 27.2 Scenario for deletion of Customer objects.

Creating and Destroying Checkout objects

Customer objects will be getting created and deleted every minute. Changes to the Checkouts are less common. Checkout objects are only added or removed when the floor manager is performing one of his/her regular checks (the problem specification suggested that these checks would be at intervals of between 5 and 60 minutes). Checkouts are added/removed in accord with the rules given earlier.

There seems to be a definite roll for a "Manager" object. It will get to "run" at regular intervals. Its "run" function will apply the rules for adding and removing checkouts.

The Manager object will need to be able to get at various data owned by the Shop object, like the number of customers present. Maybe class Manager should be a friend of class Shop, otherwise class Shop is going to have to provide a large set of access functions for use by the Manager object.

Figure 27.3 illustrates ideas for scenarios involving the creation and deletion of checkout objects. These operations will occur in (some function called) from Manager::Run(). The overall processing in Manager::Run() will start with some interactions between the Manager object and the Shop object. These will give the Manager the information needed to run the rules for opening/closing checkouts.

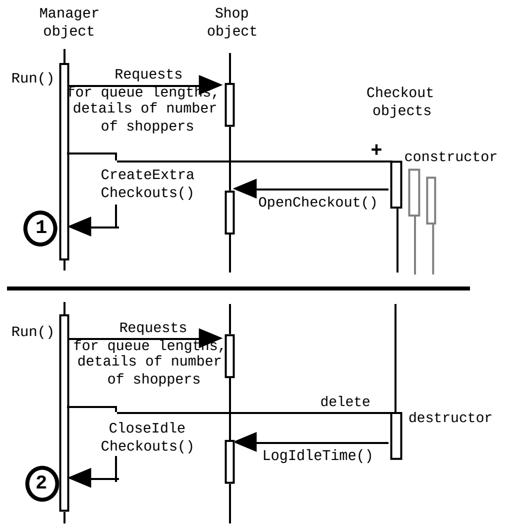


Figure 27.3 Scenarios for creation and deletion of Checkout objects.

The Shop might be involved in further interactions with existing checkouts to find their queue lengths etc; we can ignore these secondary interactions until later.

Pane 1 of Figure 27.3 illustrates an idea as to events when the shop is getting busier. The Manager will create additional checkouts. As part of their "constructor" operations, these checkout objects can register with the Shop (it will need to put them into lists and may have to perform other operations).

Pane 2 of Figure 27.3 shows the other case where there are idle checkouts that should be deleted. The Manager object will have to get access to a list of idle checkouts that would presumably be kept by the Shop object. It could then remove one or more of these, deleting the Checkout objects once they had been removed.

The destructor of the Checkout object would have to pass information to the Shop object so that it could maintain details of idle times and so forth.

***Expanded role for destructors***

Note the use of destructors in this example, and the preceding case involving Customer objects. Normally, we've used destructors only for tidying up operations. But sometimes, as here, a destructor should involve sending a notification to another object.

## Other acts of creation and destruction

The other objects appear all to be long lived. The simulation could probably start with the main() function creating the principal object which appears to be the Shop:

```
int main()
{
    // Some stuff to initialize random number generator

    Shop   aSuperMart;
    aSuperMart.Setup();
    aSuperMart.Run();
    return 0;
}
```

The Shop object could create the Door, and the Manager objects, either as part of the work of its constructor or as some separate Setup() step. Things like the PriorityQueue and Histogram objects could be ordinary data members of class Shop and so would not need separate creation steps.

The Shop, Door, and Manager objects can remain in existence for the duration of the program.

## Scenarios related to the main Run() loop in the simulation

The main Shop::Run() loop working with the priority queue will have to be something like the following:

```
while Priority Queue is not empty
    remove first thing from priority queue
    move time forward to time at which this thing is
           supposed to run
```

```
          let the thing run

          check status of thing
          if terminated
                 delete it
          if idle
                 ignore it
          if running
                 renter into priority queue

      report final statistics
```

***Need for a class hierarchy***

The priority queue contains anything that wants a chance to "run", and so it includes the Door object, a Manager object, some Checkout objects and some Customer objects. But as far as this part of the simulation is concerned, these are all just objects that can "run", can specify their "ready time", and can be asked their status (terminated, idle, running).

A simulation using the priority queue mechanism depends on our being able to treat the different objects in the priority queue as if they were similar. Thus here we are required to employ a class hierarchy. We need a general abstraction: class "Activity".

***class Activity***

Class Activity is an abstraction that describes an Activity object as something that can:

- Run()
  An Activity's Run() function will complete work on a current task and select the next task. This will be a pure virtual function. Specialized subclasses of class Activity define their own task agendas.

- Status()
  An Activity can report its status (value will be an enumerated type). Its status is either "running" (the Activity is in, or should be added to, the main priority queue used by the simulation), or "idle" (the Activity is on some "idle" list, some other object may invoke a specialized member function that change the Activity 's status), or "terminated" (the main simulation loop should get rid of those Activity objects that report they have terminated).

- Ready_At()
  An Activity can report when it is next going to be ready; the result will be in simulation time units (in this example, these units will be minute times during the day).

The current hierarchy of Activity classes is illustrated in Figure 27.4.

The scenarios shown in Figures 27.1 and 27.3 related to Run() member functions of two of the classes. The other activities triggered from this main loop will involve actions by Customer objects and Checkout objects.

A Customer object should be initially scheduled so that it "runs" when it completes its shopping phase. At this time, it should choose the Checkout where it wants to queue.

Probably, a Customer object should ask the Shop to place it on the shortest suitable queue; see Figure 27.5. This would avoid having the Customer objects interacting directly with the Checkout objects. Once it is in a Checkout queue, the Customer object can remove itself from the main simulation (it has nothing more to do); to achieve this, it just has to set its state to "idle".

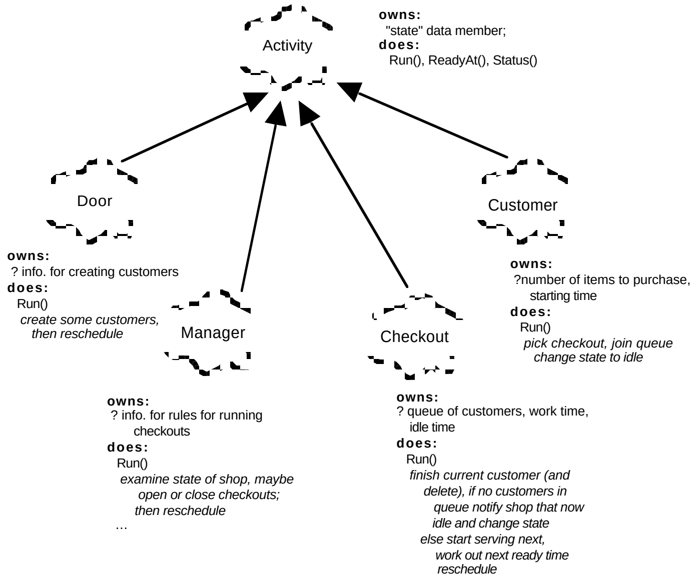

Activity
owns:
"state" data member;
does:
Run(), ReadyAt(), Status()
Door
owns:
? info. for creating customers
does:
Run()
create some customers, then reschedule
Manager
owns:
? info. for rules for running checkouts
does:
Run()
examine state of shop, maybe open or close checkouts;
then reschedule
…
Checkout
owns:
? queue of customers, work time, idle time
does:
Run()
finish current customer (and delete), if no customers in queue notify shop that now idle and change state
else start serving next, work out next ready time reschedule
Customer
owns:
?number of items to purchase, starting time
does:
Run()
pick checkout, join queue
change state to idle

Figure 27.4 A hierarchy of "Activity" classes.

Checkout objects are scheduled to "run" at the time they finish processing a current customer. At this time they can delete the customer (scenario in Figure 27.2). They then have to check their queues. If a Checkout's queue is empty, it should set its state to "idle" and inform the Shop (this needs to keep track of idle checkouts); once it is idle, the Checkout drops out of the simulation but can be reinserted if the Shop object gives the Checkout more work and gets it to reschedule itself. A Checkout should note the time that it becomes idle so that it can report its idle time. Of course, Checkouts will usually find other Customers queuing; the first Customer should be removed from the queue for processing. The Checkout can calculate its next "ready at" time from the amount of the Customer's purchases and will continue in the "running" state.

Checkout objects and Customer objects drop out of the simulation loop by becoming "idle"; some other interaction causes them to be deleted. The simulation continues until the queue is empty. The Customer and Checkout objects get removed as they become idle, but what about the Door and the Manager.

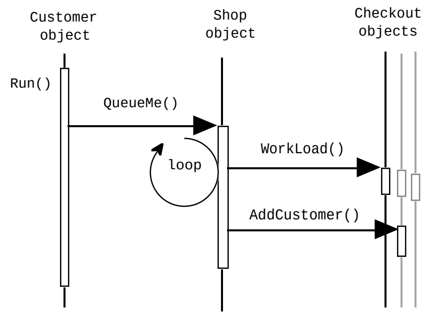


Figure 27.5 Interactions with Customer::Run().

These two normally reschedule themselves as running; the Door object setting its next "ready at" time for one minute later, the Manager selecting a time based on the frequency of checks. The Manager is responsible for getting the Shop to close its Door sometime after 7 p.m.. This is probably the basis for getting the Door removed from the simulation. If it has been "closed", instead of rescheduling itself the Door should report that it has "terminated". The simulation loop will then get rid of it.

Similarly, the Manager object should "terminate" once all its work has been done. This will be when the door has been closed and the last customer has been finished. The termination of the Manager will leave the priority queue empty and the simulation will stop.

When the main simulation loop ends, the Shop can printout the statistics that it has gathered.

## 27.2.3 Filling out the definitions of the classes for a partial implementation

The analysis of the problem is still incomplete. We haven't yet really covered how the statistics are gathered or how the histograms are displayed; nor have the rules for opening and closing checkouts been considered in any detail.

Despite that, it would be reasonable to implement a simplified version of the overall simulation at this stage. Such a partial implementation makes it possible to fully test some of the classes, e.g. class Door, and clarify other aspects such as the use of the priority queue in the simulation.

***Disadvantages of partial, prototype implementations***

Of course, some of the code created for a partial implementation is usually irrelevant to the final program. You may have to have extra functions that do, in a simplified way, some work that will later be the responsibility of a different object. You may assign responsibility for some data to one class, only to find later that you have to move those data elsewhere. Some code is thrown away. Some time is wasted.

*Advantages of partial, prototype implementations*

Although there are disadvantages associated with partial "prototype" implementations they do have benefits. If you "analyse a little, design a little, and implement a little", you get a better understanding of the problem. Besides, you may find it boring to have to work out everything on paper in advance; sometimes it is "fun" to dive in and hack a little.

The simplified version of the program could omit the `Checkouts`. `Customer` objects would just change their state to "terminated" when they finish shopping. The `Manager`'s role can be restricted to arranging for regular reports to be produced; these would show the number of customers present. Such changes eliminate most of the complexities but still allow leave enough to allow testing of the basic simulation mechanisms.

*Classes to be defined*

The classes needed in the reduced version are: `Door`, `Manager`, `Customer` (all of which are specialized subclasses of class `Activity`) and `Shop`.

*Reused classes*

In addition, the `Shop` will use an instance of class `PriorityQ` and possibly some instances of class `List`. These will simply be the standard classes from Chapter 24. The priority queue needs one change from the example given there; the maximum size of the queue should now be 1000 rather than 50 (change the constant `k_PQ_SIZE` in the class header). The `List` class should not need any changes.

*Class Activity*

The abstract class, class `Activity`, should be defined first:

```
class Activity {
public:
    enum State { eTERMINATED, eIDLE, eRUNNING };
    Activity(State s = eRUNNING) : fState(s) { }
    virtual ~Activity() {}
    virtual void  Run() = 0;
    virtual long  Ready_At() = 0;
    State         Status() { return fState; }
protected:
    State  fState;
};
```

*Pure virtual functions to be defined in subclasses*

It is simply an interface. `Activity` objects are things that "run" and report when they will next be ready; how they do this is left for definition in subclasses. Functions `Run()` and `Ready_At()` are defined as pure virtual functions; the subclasses must provide the implementations.

*Definitions for other member functions*

The abstract class can however define how the `Status()` function works as this function is simply meant to give read access to the `fState` variable. The constructor sets the `fState` data member; the default is that an `Activity` is "running". Data member `fState` is protected; this allows it to be accessed and changed within the functions defined for subclasses.

*virtual destructor*

Note the virtual destructor. Objects are going to be accessed via `Activity*` pointers; so we will get code like:

```
    Activity* a = (Activity*) fSim.First();
    …
    switch (a->Status()) {
case  Activity::eTERMINATED:
                  delete a;
                  break;
```

The `delete a` operation has to cause the appropriate destructor to be executed. If you did not specify `virtual ~Activity()` (e.g. you either didn't include a destructor or you declared it simply as `~Activity()`) the compiler would assume that it was sufficient either to do nothing special when `Activity` objects were deleted, or to generate code that included a call to `Activity::~Activity()`. However, because `virtual` has been specified correctly, the compiler know to put in the extra code that uses the table lookup mechanism (explained in Chapter 26) and so at run-time the program determines whether a call should be made to `Customer:: ~Customer()`, or `Manager::~Manager()`, or `Door::~Door()`.

If a subclass has no work to be done when its instances are deleted, it need not define a destructor of its own. (A definition has to be given for `Activity:: ~Activity()`; it is just an empty function, `{ }`, because an `Activity` has nothing of its own to do.)

*Class Door*

Class `Door` must provide implementations for `Run()` and `Ready_At()`. It will obviously have its own constructor but it isn't obvious that a `Door` has to do anything when it is deleted so there may not be a `Door::~Door()` destructor (the "empty" `Activity::~Activity()` destructor function will get used instead). Apart from `Run()`, `Status()`, and `Ready_At()`, the only other thing that a `Door` might be asked to do is `Close()`.

The specification implies that a file already exists with definitions of the arrays of arrival times and purchase amounts. These are already defined as "filescope" variables. If you had the choice, it would be better to have these arrays as static data members of class `Door`. However, given the specification requirements, the file with the arrays should just be #included into the file with the code file Door.cp. The `Door` functions can just use the arrays even though they can't strictly be "owned" by class `Door`.

The other data that a `Door` needs include: a pointer to the `Shop`, a flag indicating whether the `Door` is "open", a counter that lets it step through the successive elements in the `Arrivals[]` and `Purchases[]` arrays, and a long integer that holds the "time" at which the `Door` is next ready to run.

The class declaration should be along the following lines:

```
class Door : public Activity {
public:
    Door(Shop* s);
    void          Close();
    virtual void  Run();
    virtual long  Ready_At() { return fready; }
private:
    Shop          *fs;          // Link to Shop object
    long          fready;       // Ready time
    short         fopen;        // Open/closed status
    short         fndx;         // Next entries to use from
                                // Arrivals[], Purchases[]
};
```

The initial part of the declaration:

```
class Door : public Activity {
```

essentially states that "A Door *is a kind of* Activity". Having seen this, the C++ compiler knows that it should accept a programmer using a `Door` wherever an `Activity` has been specified.

The behaviours for the functions are:

Constructor
Set pointer to `Shop`; initialize `fready` with time from `Shop` object; set `fopen` to true; and `fndx` to 0.
(The `Shop` object will create the `Door` and can therefore arrange to insert it into the priority queue used by the simulation.)

Run()
If `fopen` is false, set `fState` (inherited from `Activity`) to "terminated".
Otherwise, pick number of arriving customers (use entry in `Arrivals[]`, get random number based on that value as default).
Loop creating `Customer` objects.

Close()
Set `fopen` to false.

This version of class `Door` should not need any further elaboration for the final program.

***class Customer***

Class `Customer` is again a specialized `Activity` that provides implementations for `Run()` and `Ready_At()`. The constructor will involve a `Customer` object interacting with the `Shop` so that the counts of `Customers` can be kept and `Customer` objects can be incorporated into the simulation mechanism.

The destructor will also involve interactions with the `Shop`; as they leave, `Customers` are supposed to log details of their total service times etc.

The only other thing that a `Customer` object might be asked is to report the number of purchases it has made. This information will be needed by the `Checkout` objects once that class has been implemented.

A `Customer` object needs a pointer to the `Shop`, a `long` to record the time that it is next ready to run, a `long` for the number of items, and two additional long integers to record the time of entry and time that it started queuing at a checkout.

A declaration for the class is:

```
class Customer : public Activity{
public:
    Customer(Shop* s, short mean);
    ~Customer();
    virtual void  Run();
    virtual long  Ready_At() { return fready; }
    long          ItemsPurchased() { return fitems; }
private:
    Shop          *fs;          // Link to shop
    long          fready;       // Ready time
    long          fstarttime;   // Other time data
    long          fstartqueue;
    long          fitems;       // # purchases
};
```

The behaviours for the functions are:

Constructor
Set pointer to `Shop`. The second argument to the constructor is to be used to pick the number of items to purchase; the value is to be an integer from an exponential distribution with the given mean value. Pick `fitems` accordingly.
Record the start time (getting the current time from the `Shop` object).
Choose a shopping rate (random in range 1…5) and use this and the value of `fitems` to calculate the time the `Customer` will be ready to queue.
Tell `Shop` to "Add Customer".

Run()
In this limited version, just set `fState` to terminated.

Destructor
Get `Shop` to log total time; then tell shop that this `Customer` is leaving.

The final program will require changes to `Run()`. The `Customer` object should get the `Shop` to transfer it to the queue at one of the existing `Checkouts` and set its own state to "idle" rather than "terminated".

***class Manager***

Class `Manager` is the third of the specialized subclasses of class `Activity`. Once again, it has to provide implementations for `Run()` and `Ready_At()`. Its constructor might be a good place to locate the code that interacts with the user to get the parameters that control the simulation. It does not appear to need to take any special actions on deletion so may not need a specialized destructor.

So far, it seems that the `Manager` will only be asked to `Run()` (and say when it is ready) so it may not need any additional functions in its public interface. Its `Run()` behaviour will eventually become quite complex (it has to deal with the rules for opening and closing checkouts). Consequently, the final version may have several private member functions.

The `Manager` object seems a good place to store most of the control parameters like the frequency of floor checks, the minimum numbers of fast and standard checkouts and the control values that trigger the opening of extra checkouts. Other information required would include the number of customers shopping and queuing at the last check time (the `Shop` object can be asked for the current numbers). The only parameter that doesn't seem to belong solely to the `Manager` is the constraint on the number of purchases allowed to users of fast checkouts. This gets used when picking queues for `Customers` and so might instead belong to `Shop`.

A declaration for the class is:

```
class Manager : public Activity{
public:
    Manager(Shop* s);
    virtual void  Run();
    virtual long  Ready_At() { return fready; }
private:
    // Will eventually need some extra functions that
    // select when to open/close checkouts

    Shop          *fs;          // Link to shop
    long          ft;           // time between checks
    long          fready;       // Ready time
    short         fmaxcheckouts;// Control parameters
    short         fminfast;
    short         fminstandard;
    short         fqlen;
    short         fqtime;
    short         fQueuingLast;
    short         fShoppingLast;
};
```

(All three classes uses the same style of implementation of the function `Ready_At()`; this suggest that it could be defined as a default in the `Activity` class itself.)

The behaviours for the functions are:

Constructor
Set pointer to `Shop`; initialize `fready` with time from `Shop` object; set `fopen` to true; and records from "last check" to -1.
(The `Shop` object will create the `Manager` and can therefore arrange to insert it into the priority queue used by the simulation.)

Prompt the user to input the values of the control parameters for the simulation.

Run()
If the time is after the shop's closing time, get the shop to make certain that the door is closed, check the number of customers still present and if zero change own status to "terminated".
Otherwise get the shop to print a status display and reschedule the `Manager` to run again after another `ft` minutes.

The final program will require changes to `Run()`. The `Manger` object will have to interact with the `Shop` to get `Checkouts` opened or closed.

***class Shop***

Class `Shop` is shaping up to be the most elaborate component in the simulation. A declaration with the members identified so far is:

```
class Shop {
public:
    Shop();
```

```
    // Organizing simulation
    void   Setup();
    void   Run();
    // Changing customers (and checkouts)
    void   AddCustomer(Customer* c);
    void   CustomerLeaves();
    // Display of state
    void   DisplayState();

    // keeping statistics
    void   LogShopTime(int);    // Customer times
    void   LogQueueTime(int);
    void   LogTotalTime(int);
    void   LogNumberPurchases(int);

    void   LogOpenTime(int);    // Checkout times
    void   LogIdleTime(int);

    // Time details
    long   Time();

    // closing
    void   CloseDoor();

    // To save having lots of access functions
    friend class Manager;
private:
    PriorityQ     fSim;

    Manager       *fManager;
    Door          *fDoor;

    long          fTime;

    short         fCustomersPresent;
    short         fCustomersQueuing;
    short         ffastlanemax;

    long          fidle;        // Idle time of checkouts
    long          fworktime;    // Total open time of checkouts

    …
};
```

The final program will have several additional data members (lists to keep records of `Checkout` objects etc) and some additional member functions.

***Use of a friend relation***

It seems worthwhile making class `Manger` a friend of class `Shop`. The `Manager` needs access to data such as the number of customers present. Rather than provide a series of access functions for all the various data elements that might be needed, we can use a friend relation. (It isn't just a program "hack"; it is appropriate that a `Manager` know all details of the `Shop`.)

The functions used to record statistical data can start to be defined, even though at this stage they may have empty implementations. Similarly, we can start to have data members that will be used to store the statistics, e.g. an integer `fIdle` to store the total time that there were checkouts that were open but idle.

The behaviours for the functions are:

***Member functions of class Shop***

Constructor
Initialize all data members (`fIdle = 0; fTime = kSTARTTIME;` etc).

Setup()
Create `Manager` and `Door` objects, insert them into the `PriorityQueue fSim`.

Run()
Implementation of loop shown earlier in which `Activity` objects get removed from the priority queue and given a chance to run.

AddCustomer()
Update count of customers present, and insert customer into priority queue.

CustomerLeaves()
Decrement counter.

DisplayState()
Show time and number of customers (might need an auxiliary private member function to "pretty print" time).
Full implementation will need display details of active and idle checkouts as well.

Log functions
All empty "stubs" in this version.

Time()
Access function allowing instances of other classes to have read access to `fTime`.

CloseDoor()
If `fDoor` pointer not `NULL`, tell door to close then set `fDoor` to `NULL`.

There doesn't appear to be any need for a destructor. Output of the time in a "pretty" format (e.g. 9.07 a.m., 3.56 p.m.) might be handled in some extra private `PrintTime()` member function.

## 27.3 A PARTIAL IMPLEMENTATION

### main() and auxiliary functions

Most of the code for `main()` was given earlier (just before introduction of the "activity" class hierarchy). There is one extra feature. The manager using the program wants to be able to run simulations of slightly different patterns of customer arrivals, and simulations using an identical pattern of arrivals but different parameters.

***Seeding the random number generator***

This can be achieved by allowing the user to "seed" the pseudo random number generator. Since "pseudo random numbers" are generated algorithmically, different runs using the same seed will get identical sequences of numbers. Use of a different seed results in a different number sequence.

```
int main(int,char**)
{

    long   aseed;
    cout << "Enter a positive integer to seed the random "
                  "number generator\n";
    cin >> aseed;

    srand(aseed);

    Shop   aSuperMart;
    …      // as shown above
```

***Random numbers from an exponential distribution***

The random number generator in the standard maths library produces numbers that are uniformly distributed. This program also requires some numbers that are taken from an exponential distribution with a defined mean (such a distribution has large numbers of small values, and a tail with large values). Most versions of the maths library don't include this version of the number generator. The required function, `erand()`, uses the normal random number generator `rand()` and a few mathematical conversions to produce random numbers with the required characteristics:

***erand()***

```
int erand(int mean)
{
    return int(-mean *log(
                  double(SHRT_MAX-rand() + 1)/SHRT_MAX )
                + 0.5);
}
```

This function can be defined in the file with the code for class `Customer`; the header files math.h, stdlib.h, and limits.h must be #included. (This implementation of `erand()` assumes that `rand()` generates values in the range `0…SHRT_MAX`. Your implementation may use a different range in its random number generator, so you may need to modify this definition of `erand()`. The range used by `rand()` is supposed to be defined by constants in the limits.h header file but many systems do not comply.)

## Module structure

*Header file dependencies*

Like the examples in Chapter 22, this program should be built from many separate files (a header file and an implementation file for essentially every class). Consequently, you have to sort out "header dependencies".

A possible arrangement of files, and most of the header dependencies, is shown in Figure 27.6. (Standard header files provided by the IDE are not shown.)

The main program defines an instance of class `Shop` and therefore must #include the Shop.h header. The class definition in Shop.h will specify that a `Shop` object contains as a `PriorityQ` as a data member. Consequently, the header Shop.h can only be handled if the header declaring the `PriorityQ` has been read; hence the dependency from Shop.h to pq.h.

The declaration of class `Shop` also mentions `Manager`, `Customer`, and `Door` but these only appear as pointer types. It isn't essential to read the declarations of these classes to compile Shop.h; but these names must be defined as class names. So file Shop.h starts with simple type declarations:

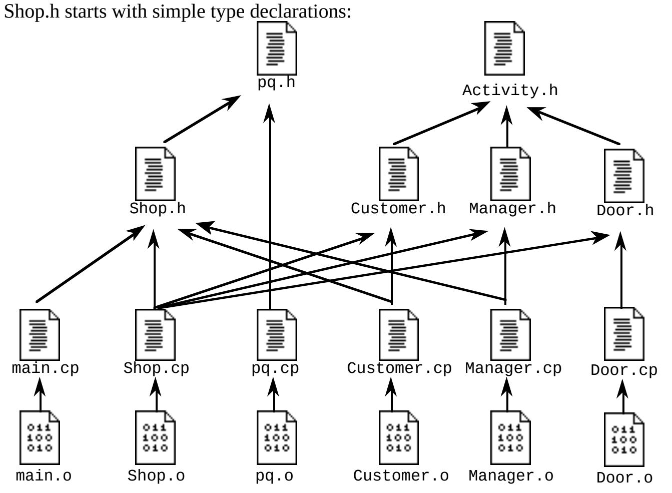


Figure 27.6 Module structure and header dependencies for partial implementation of "Supermarket" example.

*Shop.h header file*

```
#ifndef __SHOP__
#define __SHOP__

#ifndef __MYPQ__
#include "pq.h"
#endif
```

```
const int kSTARTTIME =  480;  // 8am, in minutes
const int kCLOSETIME =  540; // 9am, in minutes

class Door;
class Manager;
class Customer;

class Shop {
public:
    Shop();
    …
    void   AddCustomer(Customer* c);
    …
    friend class Manager;
private:
    PriorityQ     fSim;
    …

    Manager       *fManager;
    Door          *fDoor;
    …
};

#endif
```

*Protect against multiple #includes*

All header files must be bracketed with `#ifdef __XX__ … #endif` conditional compilation directives to avoid problems from multiple inclusion. (You can see from Figure 27.6, that the file "Activity.h" will in effect be #included three times by Shop.cp.) The italicised lines in the file listing above illustrate these directives.

*Reducing compile times*

By now you must have noticed that the majority of time used by your compiler is devoted to reading header files (both Symantec and Borland IDE's have compilation-time displays that show what the compiler is working on, just watch). Having #ifdef … #endif compiler directives in the files eliminates errors due to multiple #includes but the compiler may still have to read the same file many times. The example above illustrates a technique that can slightly reduce compile times; you will note that the file "pq.h" will only get opened and read if it has not already been read.

*The constants kSTARTTIME …*

The two constants define the start and end times for the partial simulation; these values are need in Manger.cp and Shop.cp. The values will be changed for the full implementation.

The implementation file Shop.cp contains calls to member functions of the various "activity classes"; consequently, Shop.cp must #include their headers (so that the compiler can check the correctness of the calls). These dependencies are shown in Figure 27.6 by the links from Shop.cp to Customer.h etc.

The three "activity classes" all need to #include Activity.h into their own header files. As they all have `Shop*` data members, their headers will need a declaration of the form class `Shop`;. Classes `Customer` and `Manager` both use features of class `Shop`, so their implementation files `have` to #include the Shop.h header.

The final implementation will add classes `List`, `Histogram`, and `Checkout`. Their files also have to be incorporated into the header dependency scheme. Class

List and Histogram have to be handled in the same way as class PriorityQ; class Checkout will be the same as class Manager.

### class Shop

The constructor for class Shop is trivial, just a few statements to zero out counters and set fTime to kSTARTTIME. The Setup() function creates the collaborating objects. Both need to have a Shop* pointer argument for their constructors that is supposed to identify the Shop object with which they work; hence the this arguments.

```
void Shop::Setup()
{
    fDoor = new Door(this);
    fManager = new Manager(this);
    fSim.Insert(fManager, fManager->Ready_At());
    fSim.Insert(fDoor, fDoor->Ready_At());
}
```

Once created, the Manager and Door objects get inserted into the PriorityQ fSim using their "ready at" times as their priorities.

The main simulation loop is defined by Shop::Run():

```
void Shop::Run()
{
    while(!fSim.Empty()) {
           Activity* a = (Activity*) fSim.First();
           fTime = a->Ready_At();
           a->Run();
           switch (a->Status()) {
case  Activity::eTERMINATED:
                  delete a;
                  break;
case  Activity::eIDLE:
                  break;
case  Activity::eRUNNING:
                  fSim.Insert(a, a->Ready_At());
                  }
           }
}
```

Adding a Customer object to the simulation is easy:

```
void Shop::AddCustomer(Customer* c)
{
    fSim.Insert(c, c->Ready_At());
    fCustomersPresent++;
}
```

The DisplayState() and auxiliary PrintTime() functions provide a limited view of what is going on at a particular time:

```
void Shop::DisplayState()
{
    cout << "------" << endl;
    cout << "Time ";
    PrintTime();
    cout << endl;
    cout << "Number of customers " << fCustomersPresent << endl;
}

void Shop::PrintTime()
{
    long   hours = fTime/60;
    long   minutes = fTime % 60;
    if(hours < 13) {
           cout << hours << ":" << setw(2) <<
                  setfill('0') << minutes;
           if(hours < 12) cout << "a.m.";
           else cout << "p.m.";
           }
    else cout << (hours - 12) << ":" << setw(2) <<
                   setfill('0') << minutes << "p.m.";
}
```

File Shop.cp has to #include the iomanip.h header file in order to use facilities like setw() and setfill() (these make it possible to print a time like seven minutes past eight as 8.07).

Most of the "logging" functions should have empty bodies, but for this test it would be worthwhile making Shop::LogNumberPurchases(int num) print the argument value. This would allow checks on whether the numbers were suitably "exponentially distributed".

### class Door

The constructor for class Door initialises its various data members, getting the current time from the Shop object with which it is linked.

The only member function with any complexity is Run():

```
void Door::Run()
{
    if(fopen) {
           int count = Arrivals[fndx];
           if(count>0) {
                  count = 1 + (rand() % (count + count));
                  for(int i=0;i<count;i++)
                         Customer*     c = new
                                Customer(fs,Purchases[fndx]);
                  }
           fndx++;
           fready = fs->Time() + 1;
           }
    else fState = eTERMINATED;
}
```

On successive calls, `Run()` takes data from successive locations in the file scope `Arrivals[]` and `Purchases[]` arrays. These data are used to determine the number of `Customer` objects to create. The first time that `Run()` gets executed after the `Close()` function, it changes the state to "terminated".

### class Manager

The constructor for class `Manager` can be used to get the control parameters for the current run of the program. Most input data are used to set data members of the `Manager` object; the "maximum number of items for fast checkout" is used to set the appropriate data member of the `Shop` object. (The `Manager` is a friend so it can directly change the `Shop`'s data member.)

```
Manager::Manager(Shop* s)
{
    fs = s; // Set link to shop
    cout << "Enter time interval for managers checks on "
                "queues (min 5 max 60)\n";
    short  t;
    cin >> t;

    if((t<5) || (t>60)) {
           t = 10;
           cout << "Defaulting to checks at 10 minute "
                       "intervals\n";
           }
    ft = t;

    cout << "Enter maximum number of checkouts (15-40) ";
    …
    fmaxcheckouts = t;

    cout << "Enter minimum number of fast checkouts (0-3) ";
    …
    …

    cout << "Enter maximum purchases for fast lane "
                "customers (5-15)\n";
    cin >> t;
    // Code to check value entered
    …
    // then copy into Shop object's data member
    fs->ffastlanemax = t;

    …

    fready = fs->Time();
    fQueuingLast = fShoppingLast = -1;
}
```

***Exploit "friendship" with Shop***

The `Run()` function will have to be elaborated considerably in the final version of the program. In this partial implementation it has merely to get the shop to print its state, then if the time is before the closing time, it reschedules the `Manager` to run again after the specified period. If it is later than the closing time, the `Shop` should be reminded to close the door (if it isn't already closed).

```
void Manager::Run()
{
    fs->DisplayState();
    if(fs->Time() < kCLOSETIME) {
           fready = fs->Time() + ft;
           }
    else {
           fs->CloseDoor();
           fready = fs->Time() + 5;
           if(fs->fCustomersPresent == 0) fState = eTERMINATED;
           }
}
```

The `Manager` object can "terminate" if it is after closing time and there are no `Customer` objects left in the store.

### class Customer

The constructor is probably the most elaborate function for this class. It has to pick the number of items to purchase and a shopping rate (change that `SHRT_MAX` to `RAND_MAX` if this is defined in your limits.h or other system header file). The shopping time is at least 2 minutes (defined as the constant `kACCESSTIME`) plus the time needed to buy items at the specified rate. This shopping time determines when the `Customer` will become ready.

The `Customer` object can immediately log some data with the `Shop`.

```
Customer::Customer(Shop* s, short t)
{
    fs = s;
    fitems = 1 + erand(t);
    while (fitems>250)
           fitems  = 1 + erand(t);
    fstarttime = fs->Time();
    double ItemRate = 1.0 + 4.0*rand()/SHRT_MAX;
    int shoptime = kACCESSTIME + int(0.5 + fitems/ItemRate);
    fready = fstarttime + shoptime;
    fs->AddCustomer(this);
    fs->LogShopTime(shoptime);
    fs->LogNumberPurchases(fitems);
}
```

The destructor arranges for the `Customer` object to "check out" of the `Shop`:

```
Customer::~Customer()
{
      fs->LogTotalTime(fs->Time()-fstarttime);
      fs->CustomerLeaves();
  }
```

### Testing

The other functions not given explicitly are all simple and you should find it easy to complete the implementation of this partial version of the program.

Traces from `LogNumberPurchases()` should gives number sequences like: 2, 2, 11, 2, 6, 3, 6, 12, 2, 6, 1, 15, 3, 6, 3, 5, 16, 6, 11, 3, 3, 19, …; pretty much the sort of exponential distribution expected. A test run produced the following output showing the state of the shop at different times:

```
  Time 8:00a.m.
  Number of customers 8
  ------
  Time 8:10a.m.
  Number of customers 8
  ------
  Time 8:20a.m.
  Number of customers 5
  ------
  Time 8:30a.m.
  Number of customers 19
  ------
  …
  Time 9:10a.m.
  Number of customers 2
  ------
  Time 9:15a.m.
  Number of customers 0
```

## 27.4 FINALISING THE DESIGN

There are two main features not yet implemented – operation of checkouts with customers queuing at checkouts, and the histograms. The histograms are easier so they will be dealt with first.

### 27.4.1 Histograms

The `Histogram` class comes from the "tasks" class library.

The "tasks" library is often included with C++ systems. Since it is about fifteen years old its code is old fashioned. (Also, it contains some "coroutine" components that depend on assembly language routines to modify a program's stack. If the tasks library is not included with your compiler, it is probably because no one converted these highly specialized assembly language routines. A "Coroutine" is a specialized kind of control structure used mainly in sophisticated forms of simulation.)

The histogram class keeps counts of the number of items (integer values) that belong in each of a set of "bins". It automatically adjusts the integer range represented by these bins. When all data have been accumulated, the contents of the bins can be displayed. Other statistics (number of items, minimum and maximum values, mean and standard deviation) are also output.

***class myhistogram***

The interface for the class is simple. Its constructor takes two character strings for labels in the final printout, and a number of integers that define things like the number of bins to be used. The `Add()` member function inserts another data item while `Print()` produces the output summary. There are several data members; these include `char*` pointers to the label strings, records of minimum and maximum values observed, sum and sum of squares (needed to calculate mean and standard deviation) and an array of "bins".

```
  // Based on code in ATT C++ "tasks" library.
  class myhistogram {
  public:
      myhistogram( char* title = "", char* units = "",
             int numbins = 16,int low = 0,int high = 16);
      void   Add(int);
      void   Print();
  private:
      int    l,r;          // total range covered
      int    binsize;      // range for each "bin"
      int    nbin;         // number of "bins"
      int    *h;           // the array of "bins" with counts
      long   sum;          // data for calculating average
      long   sqsum;        //     standard deviation etc
      short  count;
      long   max;          // nax and min values recorded
      long   min;
      char   *atitle;      // pointers to labels
      char   *aunits;
  };
```

The constructor works out the number of bins needed, allocates the array, and performs related initialization tasks. (It simply stores pointers to the label strings, rather than duplicating them; consequently the labels should be constants or global variables.)

```
  myhistogram::myhistogram(char* title, char* units,
      int numbins, int low, int high)
  {
      atitle = title;
      aunits = units;

      int i;
      if (high<=low || numbins<1) {
             cerr << "Illegal arguments for histogram\n";
             exit(1);
             }
      if (numbins % 2) numbins++;

      while ((high-low) % numbins) high++;
```

***Allocate the bins array***

```
        binsize = (high-low)/numbins;
        h = new int[numbins];

        for (i=0; i<numbins; i++) h[i] = 0;
        l = low;
        r = high;
        nbin = numbins;
        sum = 0;
        sqsum = 0;
        count = 0;
        max = LONG_MIN;
        min = LONG_MAX;
    }
```

The Add() member function does quite a lot more than just increment a counter. Most of the code concerns recalculation of the total range represented and size of each bin in accord with the data values entered. The various counters and sums etc are also updated.

```
    void myhistogram::Add(int a)
    {
        count++;

        max = (max > a) ? max : a;
        min = (min < a) ? min : a;

        /* add a to one of the bins, adjusting histogram,
        if necessary */
        int i, j;
        /* make l <= a < r, */
        /* possibly expanding histogram by doubling binsize
        and range */
        while (a<l) {
                l -= r - l;
                for (i=nbin-1, j=nbin-2; 0<=j; i--, j-=2)
                        h[i] = h[j] + h[j+1];
                while(i >= 0) h[i--] = 0;
                binsize += binsize;
                }
        while (r<=a) {
                r += r - l;
                for (i=0, j=0; i<nbin/2 ; i++, j+=2)
                        h[i] = h[j] + h[j+1];
                while (i < nbin) h[i++] = 0;
                binsize += binsize;
                }
        sum += a;
        sqsum += a * a;
        h[(a-l)/binsize]++;
    }
```

The Print() function outputs the summary statistics and then loops through the array of bins outputting their values as numbers.

```
    void myhistogram::Print()
    {
        if(count <= 1) {
                cout << "Too little data for histogram!\n";
                return;
                }

        cout << "\n\n" << atitle << "\n\n";
        cout << "Number of samples " << count << "\n";

        double average = ((double)sum)/count;
        double stdev =
                sqrt(((double)sqsum - sum*average)/(count-1));

        cout << "Average = " << average << aunits  << "\n";
        cout << "Standard deviation = " << stdev << aunits <<
    "\n";

        cout << "Minimum value = " << min << aunits << "\n";
        cout << "Maximum value = " << max << aunits << "\n";

        int    i;
        int    x;
        int d = binsize;

        for (i=0; i<nbin; i++) {
                x = h[i];
                if (x != 0) {
                        int ll = l+d*i;
                        cout << "[";
                        cout << setw(4) << setfill(' ');
                        cout << ll << " :" << ll + d << "]\t: ";
                        cout << setw(6) << x << "\n";
                }
        }
    }
```

The function uses features from the iomanip library so the iomanip.h header file must be #included.

***Use of histograms***

The program is supposed to produce histograms showing the variations in numbers of items purchased, queuing times, shopping times, and total times spent by customers. Consequently, class Shop had better have myhistogram data members for each of these profiles. (The statistics needed for idle times of checkouts etc just require totals rather than histograms).

```
    class Shop {
    public:
        …
    private:
        …
        // As before, plus
        myhistogram   fPurchases;
        myhistogram   fQTimes;
        myhistogram   fShopTimes;
        myhistogram   fTotalTimes;
```

```
};
```

*Changed constructor for class Shop*

The constructor for class `Shop` has to arrange for the initialization of its `myhistogram` data members. The constructor for the `myhistogram` class has defaults for its arguments, but we would want the histograms to have titles so explicit initialization is required:

```
Shop::Shop() : fPurchases("Number items purchased"),
    fQTimes("Queuing times"), fShopTimes("Shopping times"),
    fTotalTimes("Total time customer in shop")
{
    fTime = kSTARTTIME;
    fidle =  fworktime = 0;
    fCustomersPresent = fCustomersQueuing = 0;
}
```

Constructors for data members have to be invoked prior to entry to the body of the constructor for the class. As previously illustrated, Chapter 25, the calls to the constructors for data members come after the parameter list for the class's constructor and before the `{` begin bracket of the function body. A colon separates the list of data member constructors from the parameter list.

*Effective implementation for the "log" functions*

The "log" functions simply request that the corresponding `myhistogram` object "add" an item to its record:

```
void Shop::LogNumberPurchases(int num)
{
    fPurchases.Add(num);
}
```

*Extra Shop:: ReportStatistics() private member function*

The statistics gathered in a run are to be printed when the main loop in `Shop::Run()` completes. Class `Shop` should define an additional private member function, `ReportStatistics()`, that gets called from `Run()` to print the final information. A preliminary implementation would be:

```
void Shop::ReportStatistics()
{
    fPurchases.Print();
    fShopTimes.Print();
}
```

*Retesting*

The existing partial implementation can be extended to test the histogram extensions. It should produce outputs like the following:

```
Number items purchased

Number of samples 234
Average = 6.371795
Standard deviation = 6.753366
Minimum value = 1
Maximum value = 47
[   0 :4]       :    100
[   4 :8]       :     72
```

```
[   8 :12]      :     29
…
…
[  44 :48]      :      1

Shopping times

Number of samples 234
Average = 4.602564
Standard deviation = 3.583485
Minimum value = 2
Maximum value = 39
[   0 :4]       :    112
[   4 :8]       :     96
…
[  36 :40]      :      1
```

## 27.4.2 Simulating the checkouts and their queues

All that remains is the checkouts. The checkouts are relatively simple in themselves; the complexities lie in the operations of `Shop` and `Manager` that involve `Checkout` objects.

### What do Checkout objects get asked to do?

`Checkout` objects will get created at the behest of the `Manager` (the `Manager` might create the `Checkout` and pass it to the `Shop`, or the `Manager` might direct the `Shop` object to handle creation). They are created as either "fast" or "standard" and don't subsequently change their type. When initially created, they are "idle".

*Adding customers*

`Checkout` objects will be asked to "add a customer" to a queue. The queue can be represented by a `List` data member; customers get appended to the end of the list and get removed from the front. The `Shop` object will be responsible for giving `Customer` objects to the `Checkout` objects. The `Shop` can take responsibility for dealing with cases where a `Customer` gets added to an idle `Checkout`; after it has dealt with the addition operation, the `Checkout` should get put into the `Shop`'s `PriorityQ fSim`. The `Checkout` object however will have to do things like noting the time for which it has been idle.

*Ready_At() and Run()*

Since a `Checkout` is a kind of `Activity`, it must provide implementations of `Run()` and `Ready_At()`. The `Ready_At()` function can be handled in the same way as was done for the other specialized `Activity` classes. The `Run()` function will get called when a `Checkout` is due to have finished with a `Customer`; that `Customer` can be deleted. If there are other `Customer` objects queued, the `Checkout` can remove the first from its list. The `Customer` should be told to report the length of time that it has been queuing. If the queue is empty, the `Checkout` should mark its state as "idle" and should notify the `Shop`.

*Queuelength and workload*

The `Manager` object's rules for creating and destroying `Checkout` objects depend on details like average queuelengths and workloads. The `Shop` object is supposed to pick the best `Checkout` for a `Customer`; choosing the best depends on details of workload and type ("fast" or "standard"). Consequently, a `Checkout` will have to be able to supply such information.

***DisplayState***

`Checkout` objects that are not idle should display their queues as part of the `Shop` object's `DisplayState()` process.

***Destructor***

When a `Checkout` object gets deleted, it should report details of its total time of operation and its idle time. These reports get made to the `Shop` object so that relevant overall statistics can be updated.

## What do Checkout objects own?

`Checkout` objects will need:

- a pointer to the `Shop` object;
- long integers representing time current task completed, time current task started, time of creation, total of idle periods so far;
- a list to hold the queuing `Customer` objects;
- a pointer to the `Customer` currently being served;
- a short to hold the "fast" or "standard" type designation.

## Design diagram for class Checkout

Figure 27.6 is a class "design diagram" that summarizes the features of class Checkout. Such diagrams are often used as a supplement to (or alternative to) textual descriptions like those given earlier for the other Activity classes like class Manager.

The diagram in Figure 27.7 is simpler than the styles that you will be using later when you have been taught the current standard documenting styles, but it still provides an adequate summary. The header should give details of inheritance. There should be a means of distinguishing the public interface and private implementation. Public functions should have a brief comment explaining their role (often details of return types and argument lists are omitted because these design diagrams can be sketched out before such fine details have been resolved).

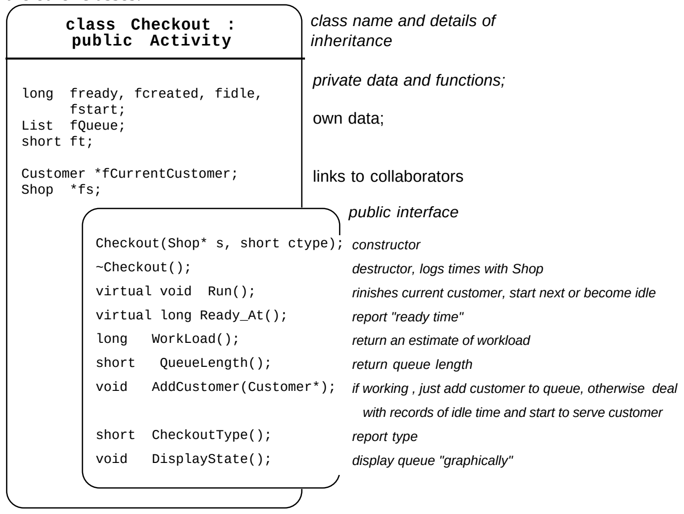
It is often useful to distinguish between "own data" and "links to collaborators". Both are implemented as "data members". It is just a matter of a different role, but it is worth highlighting such differences in documentation.

## Implementation of class Checkout

The implementation of class `Checkout` will identify a few extra member functions that will be needed in class `Customer` and class `Shop`.

For example, a `Checkout` has to notify a `Customer` when processing starts; this is needed to allow the `Customer` object to work out its queuing time and report this to the `Shop`. Consequently, class `Customer` will need an additional function, "finish queuing", in its public interface.

Such extra functions would be noted while the implementation of class `Checkout` was sketched out. Subsequently, the extra functions would be defined in the other classes.



```
class Checkout :
public Activity

long  fready, fcreated, fidle,
      fstart;
List  fQueue;
short ft;

Customer *fCurrentCustomer;
Shop  *fs;

Checkout(Shop* s, short ctype);
~Checkout();
virtual void  Run();
virtual long Ready_At();
long   WorkLoad();
short   QueueLength();
void   AddCustomer(Customer*);

short  CheckoutType();
void   DisplayState();
```

class name and details of inheritance
private data and functions;
own data;
links to collaborators
public interface
constructor
destructor, logs times with Shop
rinishes current customer, start next or become idle
report "ready time"
return an estimate of workload
return queue length
if working , just add customer to queue, otherwise deal with records of idle time and start to serve customer
report type
display queue "graphically"

Figure 27.7 Design diagram for class Checkout.

***Checkout's constructor***

Previous examples have shown cases where data members that were instances of classes had to have their constructors invoked prior to entry to the main body of a constructor. This time, we need to invoke the base class's constructor (`Activity::Activity()`). The other specialized subclasses of `Activity` relied on the default parameters to initialize the "activity" aspect of their objects. But, by default, the initializer for class `Activity` creates objects whose state is "running". Here we want to create an object that is initially "idle". Consequently, we have to explicitly invoke the constructor with the appropriate `eIDLE` argument; while we are doing that we might as well initialize some of the other data members as well:

```
Checkout::Checkout(Shop* s, short t) : Activity(eIDLE),
     fs(s) , ft(t)
{
    fCurrentCustomer = NULL;
    fidle = 0;
    fcreated = fready = fs->Time();
}
```

The "type" of a `Checkout` (argument t for constructor, data member `ft`) could use some integer coding (e.g. 1 => fast, 0 => standard).

*Destructor*

The destructor should finalize the estimate of idle time and then log the Checkout's idle and open times with the Shop:

```
Checkout::~Checkout()
{
    fidle += fs->Time() - fready;
    fs->LogIdleTime(fidle);
    fs->LogOpenTime(fs->Time() - fcreated);
}
```

The Run() function starts by getting rid of the current customer (if any). Then, if there is another Customer in the queue, this object is removed from the queue to become the current customer, is notified that it has finished queuing, and a new completion time calculated based on the processing rate (kSCANRATE) and the number of items purchased.

```
void Checkout::Run()
{
    if(fCurrentCustomer != NULL) {
           delete fCurrentCustomer;
           fCurrentCustomer = NULL;
           }

    if(fQueue.Length() > 0) {
           fCurrentCustomer = (Customer*) fQueue.Remove(1);
           fCurrentCustomer->FinishQueuing();
           fstart = fs->Time();
           short t = fCurrentCustomer->ItemsPurchased();
            t /= kSCANRATE;
           t = (t < 1) ? 1 : t;
           fready = fstart + t;
           }
    else  {
           fState = eIDLE;
           fs->NoteCheckoutStopping(this);
           }

}
```

Alternatively, if its queue is empty, the Checkout becomes idle and notifies the Shop of this change.

*Using a ListIterator*

The version of List used for this example could be that given in Chapter 26 with the extra features like a ListIterator. The Checkout::WorkLoad() function can use a ListIterator to run down the list of queuing customers getting each to state the number of items purchased. Unscanned items from the current customer should also be factored into this work load estimate.

*ListIterator*

```
long Checkout::WorkLoad()
{
    long res = 0;
    ListIterator LL(&fQueue);
    LL.First();
```

```
    while(!LL.IsDone()) {
              Customer *c = (Customer*) LL.CurrentItem();
              res += c->ItemsPurchased();
              LL.Next();
              }
    if(fCurrentCustomer != NULL) {
           long items = fCurrentCustomer->ItemsPurchased();
           long dlta = (items - kSCANRATE*(fs->Time()-fstart));
           dlta = (dlta < 0) ? 0 : dlta;
           res += dlta;
           }
    return res;
}
```

The work involved in adding a customer depends on whether the Checkout is currently idle or busy. Things are simple if the Checkout is busy; the new Customer is simply appended to the list associated with the Checkout. If the Checkout is idle, it can immediately start to serve the customer; it should also notify the Shop so that the records of "idle" and "busy" checkouts can be updated.

```
void Checkout::AddCustomer(Customer* c)
{
    if(fState == eIDLE) {
           fidle += fs->Time() - fready;
           fCurrentCustomer = c;
           fCurrentCustomer->FinishQueuing();
           fstart = fs->Time();
           short t = fCurrentCustomer->ItemsPurchased();
           t /= kSCANRATE;
           t = (t < 1) ? 1 : t;
           fready = fstart + t;
           fState = eRUNNING;
           fs->NoteCheckoutStarting(this);
           }
    else fQueue.Append(c);
}
```

The final DisplayState() function just has to provide some visual indication of the number of customers queuing, e.g. a line of '*'s.

### 27.4.3 Organizing the checkouts

The addition of Checkout objects in the program requires only a minor change in class Customer. Its Customer::Run() function no longer sets its state to terminated, instead a Customer should set its state to eIDLE and ask the Shop object to find it a Checkout where it can queue.

However, classes Shop and Manager must have major extensions to allow for Checkouts.

***Additional responsibilities for class Shop***

The Manager object will be telling the Shop to add or remove Checkouts; the type (fast or standard) will be specified in these calls:

```
void       Shop::AddCheckout(short ctype);
void       Shop::RemoveIdleCheckout(short ctype);
```

Checkout objects notify the Shop when the start or stop work:

```
void       Shop::NoteCheckoutStarting(Checkout* c);
void       Shop::NoteCheckoutStopping(Checkout* c);
```

and, as just noted, Customer objects will be asking the Shop to move them onto queues at Checkouts:

```
void       Shop::QueueMe(Customer* cc);
```

***Extra data members***

The Shop has to keep track of the various Checkouts, keeping idle and busy checkouts separately. It would probably be easiest if the Shop had four List objects in which it stored Checkouts:

```
List       fFastIdle;
List       fStandardIdle;
List       fFastWorking;
List       fStandardWorking;
```

The AddCheckout(), RemoveIdleCheckout() will both involve "idle" lists; the ctype argument can identify which list is involved.

When a Checkout notifies the Shop that it has stopped working, the Shop can move that Checkout from a "working" list to an "idle" list (the Shop can ask the Checkout its type and so determine which lists to use). Similarly, when a Checkout starts, the Shop can move it from an "idle" list to a "working" list (and also get it involved in the simulation by inserting it into the priority queue as well).

The only one of these additional member functions that is at all complex is the Checkout::QueueMe() function which is outlined below.

***Manager exploits friend relation***

The Manager object will need information on the numbers of idle and busy checkouts etc. Since the Manager is a friend of class Shop, it can simply make calls like:

```
long num_working = fs->fFastWorking.Length() +
                         fs->fStandardWorking.Length();
long num_idle = fs->fFastIdle.Length() +
                         fs->fStandardIdle.Length();
```

Here, the Manager uses its Shop* pointer fs to access the Shop object, exploiting its friend relation to directly use a private data member like fFastWorking; the List::Length() function is then invoked for the chosen List object.

The additional responsibilities for class Manager all relate to the application of those rules for choosing when to open and close checkouts. The Manager::Run() function needs to get these rules applied each time a check on the shop is made; this function now becomes something like:

```
void Manager::Run()
{
```

```
        fs->DisplayState();
        if(fs->Time() < kCLOSETIME) {
               Sortoutcheckouts();
               fready = fs->Time() + ft;
               }
        else {
               fs->CloseDoor();
               Closingcheckouts();
               fready = fs->Time() + 5;
               if(fs->fCustomersPresent == 0)
                      fState = eTERMINATED;
               }
    }
```

There are two additional calls to new functions that will become extra private member functions of class Manager. The function Sortoutcheckouts() will apply the full set of rules that apply during opening hours; the Closing-checkouts() function will apply the simpler "after 7 p.m. rule".

A function like Closingcheckouts() is straightforward. The Manager can get details of the numbers of checkouts (fast and standard, idle and working) by interrogating the List data members of the Shop. Using these data, it can arrange to close checkouts as they become idle (making certain that some checkouts are left open so long as there are Customers still shopping):

```
void Manager::Closingcheckouts()
{
    /* It is after 7pm, close the fast checkouts as they
    become idle. */
    int fastidle = fs->fFastIdle.Length();
    for(int i=0; i < fastidle; i++)
           fs->RemoveIdleCheckout(kFAST);

    int standardidle = fs->fStandardIdle.Length();
    int standardworking = fs->fStandardWorking.Length();

    /*
    So long as there are checkouts working can get rid of
    all idle ones (if any).
    */
    if((standardworking > 0) && (standardidle > 0)) {
           for(i=0; i < standardidle; i++)
                  fs->RemoveIdleCheckout(kSTANDARD);
           return;
           }

    /*
    If there are no customers left close all idle checkouts
    */
    if(fs->fCustomersPresent == 0) {
           for(i=0; i < standardidle; i++)
                  fs->RemoveIdleCheckout(kSTANDARD);
           return;
           }

    /*
    May end up with state where there are customers still
    shopping but all checkouts idle.  Close all but one.
    */
    for(i=0; i < standardidle - 1; i++)
            fs->RemoveIdleCheckout(kSTANDARD);
}
```

***Use "top down functional decomposition" for complex functions***

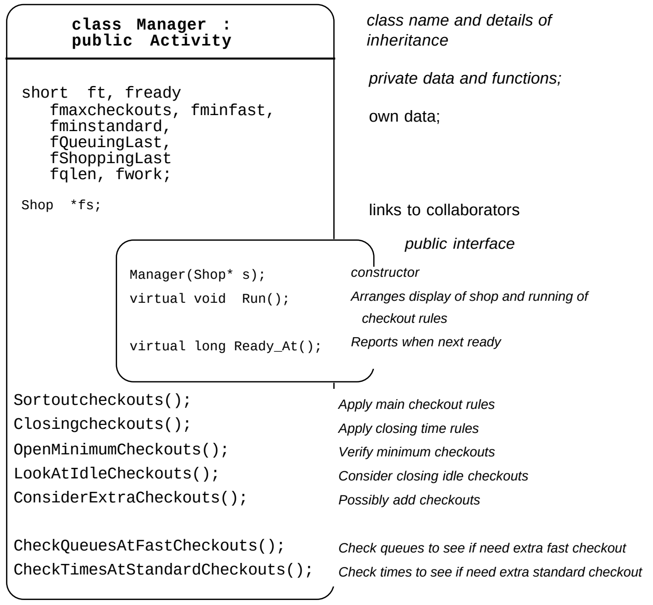
The rules that define normal operation are too complex to be captured in a single function. Once again, the functional decomposition approach has to be applied. The `Manager` has to consider three aspects: checking that minimum numbers of checkouts are open, checking for idle checkouts that could be closed, and checking for excessive queues that necessitate opening of checkouts. Inevitably, the `Sortoutcheckouts()` function becomes:

```
void Manager::Sortoutcheckouts()
{
    OpenMinimumCheckouts();
    LookAtIdleCheckouts();
    ConsiderExtraCheckouts();
}
```

The function has been decomposed into three simpler functions that allow individual consideration of the different rules. In some cases, further decomposition into yet simpler functions is necessary. Aspects of these functions are illustrated below.

## Final class definitions

Figures 27.8 and 27.9 show the finalized design diagrams for classes `Manager` and `Shop`.



```
class  Manager  :
public  Activity

short  ft, fready
   fmaxcheckouts, fminfast,
   fminstandard,
   fQueuingLast,
   fShoppingLast
   fqlen, fwork;

Shop  *fs;

Manager(Shop* s);
virtual void  Run();

virtual long Ready_At();

Sortoutcheckouts();
Closingcheckouts();
OpenMinimumCheckouts();
LookAtIdleCheckouts();
ConsiderExtraCheckouts();

CheckQueuesAtFastCheckouts();
CheckTimesAtStandardCheckouts();
```

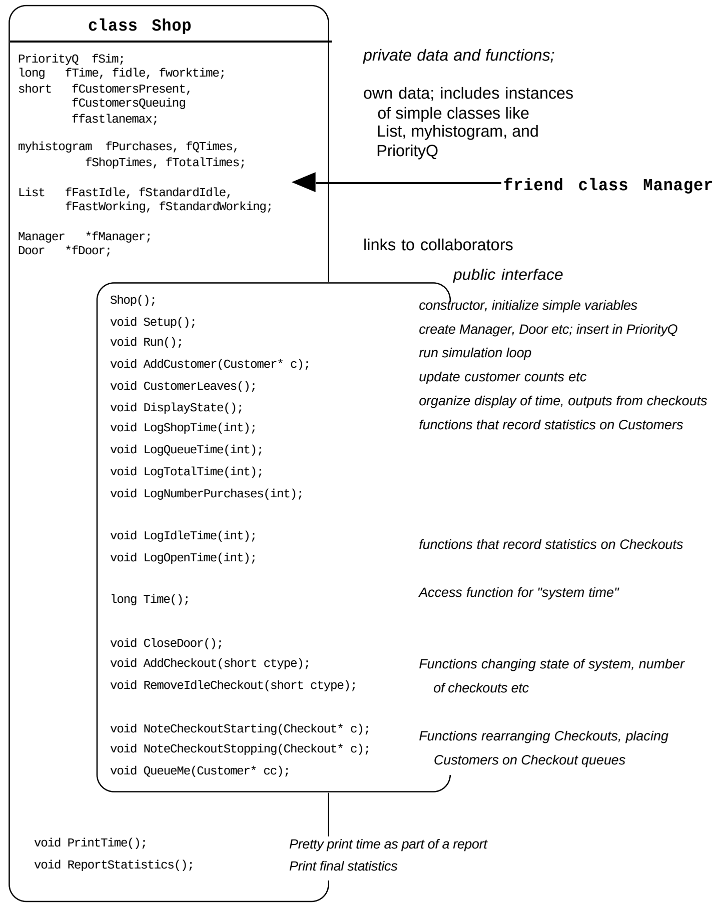
class name and details of inheritance
private data and functions;
own data;
links to collaborators
public interface
constructor
Arranges display of shop and running of checkout rules
Reports when next ready
Apply main checkout rules
Apply closing time rules
Verify minimum checkouts
Consider closing idle checkouts
Possibly add checkouts
Check queues to see if need extra fast checkout
Check times to see if need extra standard checkout

Figure 27.8 Final design diagram for class Manager.

Class `Manager` retains a very simple public interface. The `Manager` object is only used in a small number of ways by the rest of the program. However, the things that a `Manager` gets asked to do are complex, and consequently the class has a large number of private implementation functions.

Class `Shop` has an extensive public interface – many other objects ask the `Shop` to perform functions. Most of these member functions are simple (just increment a count, or move the requestor from one list to another); consequently, there is only limited need for auxiliary private member functions.

Diagram 27.9 also indicates that a "friend" relationship exists that allows a `Manager` object to break the normal protection around the private data of the `Shop` object.



```
class Shop

PriorityQ  fSim;
long   fTime, fidle, fworktime;
short   fCustomersPresent,
        fCustomersQueuing
        ffastlanemax;

myhistogram  fPurchases, fQTimes,
          fShopTimes, fTotalTimes;

List   fFastIdle, fStandardIdle,
       fFastWorking, fStandardWorking;

Manager   *fManager;
Door   *fDoor;

Shop();
void Setup();
void Run();
void AddCustomer(Customer* c);
void CustomerLeaves();
void DisplayState();
void LogShopTime(int);
void LogQueueTime(int);
void LogTotalTime(int);
void LogNumberPurchases(int);

void LogIdleTime(int);
void LogOpenTime(int);

long Time();

void CloseDoor();
void AddCheckout(short ctype);
void RemoveIdleCheckout(short ctype);

void NoteCheckoutStarting(Checkout* c);
void NoteCheckoutStopping(Checkout* c);
void QueueMe(Customer* cc);

void PrintTime();
void ReportStatistics();
```

private data and functions;
own data; includes instances of simple classes like List, myhistogram, and PriorityQ
friend class Manager
links to collaborators
public interface
constructor, initialize simple variables
create Manager, Door etc; insert in PriorityQ
run simulation loop
update customer counts etc
organize display of time, outputs from checkouts
functions that record statistics on Customers
functions that record statistics on Checkouts
Access function for "system time"
Functions changing state of system, number of checkouts etc
Functions rearranging Checkouts, placing Customers on Checkout queues
Pretty print time as part of a report
Print final statistics

Figure 27.9 Final design diagram for class Shop.

## Implementation of the more elaborate functions

The function `Shop::QueueMe(Customer*)` involves finding an appropriate `Checkout` for the requesting `Customer` object.

The code starts by determining whether the Customer is able to use both fast and standard `Checkouts`, or just the standard `Checkouts`.

Next, the function checks for idle `Checkouts` (both types if the customer is allowed to used the fast lanes, otherwise just the "idle standard" `Checkouts`). If there is a suitable idle `Checkout`, the `Customer` can be queued and the function finishes:

```
void Shop::QueueMe(Customer* cc)
{
    int fast = (cc->ItemsPurchased() <= ffastlanemax) ||
           (99 == (rand() % 100));

    /*
    If appropriate, try for an idle checkout
    */
    if(fast && (fFastIdle.Length() > 0)) {
           Checkout *c = (Checkout*) fFastIdle.Nth(1);
           c->AddCustomer(cc);
           return;
           }

    if(fStandardIdle.Length() > 0) {
           Checkout *c = (Checkout*) fStandardIdle.Nth(1);
           c->AddCustomer(cc);
           return;
           }
```

If there is no idle `Checkout`, the function has to search through either both lists of working `Checkouts` (or just the list of standard checkouts) to find the one with the least load. These searches through the lists can again take advantage of `ListIterators`:

```
    long   least_load = LONG_MAX;
    Checkout *best = NULL;

    if(fast) {
           ListIterator L1(&fFastWorking);
           L1.First();
           while(!L1.IsDone()) {
                  Checkout *c = (Checkout*) L1.CurrentItem();
                  long load = c->WorkLoad();
                  if(load < least_load) {
                         least_load = load;
                         best = c;
                         }
                  L1.Next();
                  }

           }

    ListIterator L2(&fStandardWorking);
    // Code similar to last ListIterator loop
```

The `Customer` choses the `Checkout` with the smallest workload (there had better be a test to verify that there was a `Checkout` where the `Customer` could queue):

```
        if(best == NULL) {
               cout << "Store destroyed by rioting customers."
                      "No checkouts open." << endl;
               exit(1);
               }
        else best->AddCustomer(cc);

        fCustomersQueuing++;
    }
```

The remaining functions of class `Manager` also involve interrogation of the various `List` data members of the associated `Shop` object; there are also several places where `ListIterators` get employed to work through all entries in one of these lists. The most complex of the rules used by the `Manager` is that relating to the opening of extra checkouts. The rule starts by saying that these should be considered if the number of customers queuing or shopping has increased. If this is the case, then an additional fast checkout can be opened if the average queue length is too long at fast lanes. Similarly, one or more additional standard checkouts is to open if the wait time is too long. Naturally, this breaks down into separate cases handled by separate functions. The `ConsiderExtraCheckouts()` function establishes the context for adding checkouts:

***If getting busier, add more checkouts***

```
    void Manager::ConsiderExtraCheckouts()
    {
        long shopping = fs->fCustomersPresent -
                             fs->fCustomersQueuing;
        long queuing = fs->fCustomersQueuing;

        if((shopping>fShoppingLast) || (queuing > fQueuingLast)) {
               CheckQueuesAtFastCheckouts();
               CheckTimesAtStandardCheckouts();
               }
        fShoppingLast = shopping;
        fQueuingLast = queuing;
    }
```

The function that adds fast checkouts is representative of the remaining functions. If identifies circumstances that preclude the opening of new checkouts (e.g. maximum number already open) and if any pertain it abandons processing:

```
    void Manager::CheckQueuesAtFastCheckouts()
    {
        /*
        if there are any idle checkouts do nothing.
        */

        int idle = fs->fFastIdle.Length() +
                      fs->fStandardIdle.Length();
        if(idle != 0)
               return;

        /*
        If don't have any fast checkouts open, can't check
```

```
        average length so do nothing.  (If number of customers
        is large, a fast checkout will have been created by
        the minimum checkouts rule.)
        */

        int num = fs->fFastWorking.Length();
        if(num == 0)
               return;

        int total = num + fs->fStandardWorking.Length();

        if(total == fmaxcheckouts)
               return;
```

The next step involves iterating through existing working checkouts gathering information needed to determine current load. Naturally, this is done with the aid of a `ListIterator`:

```
        int queuing = 0;
        ListIterator L1(&(fs->fFastWorking));
        L1.First();
        while(!L1.IsDone()) {
               Checkout *c = (Checkout*) L1.CurrentItem();
               queuing += c->QueueLength();
               L1.Next();
               }

        int average = queuing / num;
```

If the average queue length is too great, an extra checkout should be added:

```
        if(average < fqlen)
               return;

        fs->AddCheckout(kFAST);
        cout << "Added a fast checkout." << endl;
    }
```

## Execution

You should find it fairly simple to complete the implementation (Exercise 1). The program should produce outputs like the following:

```
Time 8:30a.m.
Number of customers 65
Fast Checkouts:
*|*************************
Standard Checkouts:
*|********************
Added a fast checkout.
Added 1 standard checkouts.
------
Time 8:40a.m.
Number of customers 84
```

```
Fast Checkouts:
*|******************
*|************
Standard Checkouts:
*|*********
*|**************
Time 9:30a.m.
Number of customers 34
There are 2 idle fast
checkouts.
Fast Checkouts:
```

```
*|
Standard Checkouts:
*|***
*|**

Closed 2 fast checkouts


Time 7:00p.m.
Number of customers 15
There are 2 idle fast
checkouts.
There are 10 idle standard
checkouts.
Standard Checkouts:
*|
*|
*|
*|
*|
*|
*|
*|
Door closed.


Shop closing at 7:35p.m.
Checkout times:
Total 8720 minutes
Idle 834 minutes

Shopping times

Number of samples 2627
Average = 14.5778
Standard deviation = 16.9764
Minimum value = 2
Maximum value = 180
[   0 :16]      :   1862
[  16 :32]      :    462
[  32 :48]      :    179
[  48 :64]      :     67
[  64 :80]      :     28
[  80 :96]      :     14
[  96 :112]     :      7
[ 112 :128]     :      1
[ 128 :144]     :      3
[ 144 :160]     :      1
```

```
…
…
Time 7:30p.m.
Number of customers 1
Standard Checkouts:
*|
------
Time 7:35p.m.
Number of customers 0
There are 1 idle standard
checkouts.

[ 160 :176]     :      2
[ 176 :192]     :      1

Queuing times

Number of samples 2627
Average = 3.24705
Standard deviation = 3.69674
Minimum value = 0
Maximum value = 23
[   0 :2]       :   1170
[   2 :4]       :    509
[   4 :6]       :    314
[   6 :8]       :    255
[   8 :10]      :    208
[  10 :12]      :     84
[  12 :14]      :     37
[  14 :16]      :     26
[  16 :18]      :     12
[  18 :20]      :      7
[  20 :22]      :      3
[  22 :24]      :      2
```

## EXERCISES

1 Complete a working version of the Supermarket program.

2 Implement a simulation of an "office information system".

The system has a number of sources of "messages" – electronic mail, a facsimile machine, a local network, etc. The different kinds of incoming messages are all placed in a single queue for processing by the executive using the system. The user can get details of the queue displayed; these details will include the number of queued messages, and a list showing the headers for each message. These header details are to include "arrival time", "sender", and "topic" (each of these is a short string). The user can select messages (identifying them by an index number) for detailed display.
Once selected, a message becomes the "current message". The system is to allow the user to display the content of the current message, requeue it, save it to file or delete it. Different message types display their content in distinct ways.

The message sources should work with data files. These text files will contain successive messages. Each message in the file starts with an "arrival time", a "sender" name, and "topic header". The content part of the message will follow; different content forms should be used for the different message types. The simulation will give message sources an opportunity to "run" at regular intervals. When a message source runs, it should read successive messages from its input file, creating appropriate message objects that get added to the main queue. Data should continue to be read, and message objects be created, until the next "arrival time" is greater than the current simulated system time.

The system is to have a simulated time scale based on actual execution time (use the functions provided in your IDE for accessing the clock). For example, six seconds of execution time could represent one minute of simulated time.

The simulation will involve a loop in which the user is prompted for a command. The program will pause until data is entered. The computer's clock is read after the user input and used to update the simulated time. All message sources are given the opportunity to "run", possibly resulting in additional messages being added to the main queue. Then the users command should be interpreted. Finally, the "simulated time" should be displayed and the user should be prompted for the next command.