Wayne Haythorn

2095 Proctor Rd, Mosier, OR 97040; email: haythorn@cse.ogi.edu

What is object-oriented design?

THIS ARTICLE IS concerned with goals and guidelines for object oriented design (OOD). Goals are high-level terms like "improve productivity" and "encourage reuse." Guidelines are more tied to the code—they answer questions like "how do I tell a good design from a bad one?"

The goals should drive the guidelines, but some goals are too vague to give us any direction. For example, almost any software method will claim to improve productivity, so "improve productivity" as a goal doesn't give us much guidance about how to code.

To find goals that are relevant to OOD, we can begin with four reasons generally given for using object-oriented methods in the first place:

1. World modeling—People see the world in terms of objects, so a program written in terms of objects should be more intuitive and understandable than a program structured in some other way.
2. Reuse—As software modules, objects have high cohesion because they encapsulate code and data. So, the chances are good that an object can be lifted from one program context and used in another.
3. Maintainability—Because objects have higher modularity, the effects of program changes should be better localized, and therefore easier and cheaper to implement.
4. Unified software method—Analysis will be done in terms of objects in the world. If design, coding, and maintenance also can be done in terms of objects, the entire software process will be governed by a single conceptual framework.

These four kinds of goals have not received equal attention. Most recent discussion and promotion of object-oriented methods has focused on the first two—world modeling and reuse. In particular, reuse has been emphasized over maintainability. I believe this is a mistake, for the following two reasons.

First, many managers will not buy the reuse argument. This is not because of the well-known fact that "managers are stupid." Management objectives should be defined within fixed time frames, but "reuse" is inherently vague in terms of the time frame for payoff. In fact, experienced object-oriented programmers frequently report that they "didn't get it right until the third time." So, it is not clear when you will produce the reusable components, much less actually reuse them. This means it is very difficult to express reuse as a meaningful business objective. Business management is oriented toward projects and products. Component-oriented management is only possible if the production and value of components is well understood.

Worse yet, the reuse goal does not lead to guidelines for system design. Reuse is a component-level concept, but what we build are systems, and components are only useful within systems. OOD requires system-level guidelines. Because reuse doesn't give us any direction, the only guideline commonly discussed in books on OOD is "model the problem domain objects." Notice how little this says about actual code. While "model the domain" helps us to identify objects, it is useless as a guideline for assigning methods to classes because the real world doesn't have to worry about instruction sequencing on a von Neumann architecture. However, because they are not clear about what makes a good program in code terms, many writers fall back on direct world modeling. Ivar Jacobson described these as the "naive" object-oriented methods.[1] It is characteristic of these methods that they are fairly convincing when we are talking about analysis (identifying objects), but they don't seem to produce a satisfactory approach to design (allocating methods and specifying message flows).

As a goal for OOD, maintainability solves both problems. It is a system-level concept with a payoff that is measurable within project time frames. Furthermore, it sells—people who are confused by talk about reuse will pay attention if you can promise reduced maintenance costs. You can make that promise, and deliver on it, because it is possible to give precise guidelines for how to make an object-oriented program maintainable.

OBJECT BASED VS. OBJECT ORIENTED

The least that an OOD method should do for us is to help us use the capabilities of object-oriented languages. These languages have

Wayne Haythorn

2095 Proctor Rd, Mosier, OR 97040; email: haythorn@cse.ogi.edu

What is object-oriented design?

THIS ARTICLE IS concerned with goals and guidelines for object oriented design (OOD). Goals are high-level terms like "improve productivity" and "encourage reuse." Guidelines are more tied to the code—they answer questions like "how do I tell a good design from a bad one?"

The goals should drive the guidelines, but some goals are too vague to give us any direction. For example, almost any software method will claim to improve productivity, so "improve productivity" as a goal doesn't give us much guidance about how to code.

To find goals that are relevant to OOD, we can begin with four reasons generally given for using object-oriented methods in the first place:

1. World modeling—People see the world in terms of objects, so a program written in terms of objects should be more intuitive and understandable than a program structured in some other way.
2. Reuse—As software modules, objects have high cohesion because they encapsulate code and data. So, the chances are good that an object can be lifted from one program context and used in another.
3. Maintainability—Because objects have higher modularity, the effects of program changes should be better localized, and therefore easier and cheaper to implement.
4. Unified software method—Analysis will be done in terms of objects in the world. If design, coding, and maintenance also can be done in terms of objects, the entire software process will be governed by a single conceptual framework.

These four kinds of goals have not received equal attention. Most recent discussion and promotion of object-oriented methods has focused on the first two—world modeling and reuse. In particular, reuse has been emphasized over maintainability. I believe this is a mistake, for the following two reasons.

First, many managers will not buy the reuse argument. This is not because of the well-known fact that "managers are stupid." Management objectives should be defined within fixed time frames, but "reuse" is inherently vague in terms of the time frame for payoff. In fact, experienced object-oriented programmers frequently report that they "didn't get it right until the third time." So, it is not clear when you will produce the reusable components, much less actually reuse them. This means it is very difficult to express reuse as a meaningful business objective. Business management is oriented toward projects and products. Component-oriented management is only possible if the production and value of components is well understood.

Worse yet, the reuse goal does not lead to guidelines for system design. Reuse is a component-level concept, but what we build are systems, and components are only useful within systems. OOD requires system-level guidelines. Because reuse doesn't give us any direction, the only guideline commonly discussed in books on OOD is "model the problem domain objects." Notice how little this says about actual code. While "model the domain" helps us to identify objects, it is useless as a guideline for assigning methods to classes because the real world doesn't have to worry about instruction sequencing on a von Neumann architecture. However, because they are not clear about what makes a good program in code terms, many writers fall back on direct world modeling. Ivar Jacobson described these as the "naive" object-oriented methods.[1] It is characteristic of these methods that they are fairly convincing when we are talking about analysis (identifying objects), but they don't seem to produce a satisfactory approach to design (allocating methods and specifying message flows).

As a goal for OOD, maintainability solves both problems. It is a system-level concept with a payoff that is measurable within project time frames. Furthermore, it sells—people who are confused by talk about reuse will pay attention if you can promise reduced maintenance costs. You can make that promise, and deliver on it, because it is possible to give precise guidelines for how to make an object-oriented program maintainable.

OBJECT BASED VS. OBJECT ORIENTED

The least that an OOD method should do for us is to help us use the capabilities of object-oriented languages. These languages have veloped and built the team will be responsible for teaching the application developers how to use the framework, ensuring that the framework is used properly, and updating the framework to support new needs.

Application view developers are responsible for defining and constructing the view classes. They do not need to know the internals of the framework but do need a good knowledge of how to build on top of it. Whilst the shared model team needs to be singular and small, the view developers can be widely distributed, responsible to small business units (see Fig. 9). This puts them closer to the users (and their needs). Communication between view developers is simplified because all such communication is carried out through the shared model.

With a firm platform of data manipulation provided by the view developers, the presentation developers can concentrate on user interface design; any programming is limited to call-backs to the application views. This, especially with modern user interface tools, is a different skill than developing the views and can be worked on by those with an aptitude for user-interface design rather than programming. It is in this role that users can play the greatest part; with the tools now available much of this work can be done by the users themselves.

In this scheme the shared model becomes the principal focus for reuse. It is in this format that shared data is accessed and any shared processing would be done in terms of the shared model. Some reuse of views and presentations would also occur, as certain blocks of information were wanted in different places. Such common elements probably would be subclassed heavily for particular users. In this way, a centralised DP department would provide both the shared model and a set of widely needed views and presentations.

OTHER USES FOR VIEWS

This discussion of application views has focused on using them for user interfaces and reports. They are also useful for other parts of the system that requires a methodical approach to simplifying a complex model. One clear use is in mapping to record-oriented databases and files, particularly relational databases. The attributes in views would correspond to attributes in a file or columns in a relational table. In this way an object-oriented shared model can be used to integrate modern user interfaces with traditional database formats.

The advantages of this approach have been noted by advocates of the three-schema architecture. Applications, based on the shared model, can be developed entirely independent of the database structure. (An analogy can be drawn here between this approach and that of portable compilers). There is, of course, a price. All mapping from database views to user interface views via the shared model clearly takes up more time than a direct user interface. The trade-off, as ever, is flexibility and ease of development vs. use of computer resources. Fortunately, the architecture does help developers to balance loads between servers and clients, to control the load on both processors and the network.

Application views also can be used to help define security, particularly in the presence of an ORB. Interfaces to a complex shared model can be presented to the ORB via application views. This would stop applications gaining unrestricted access to objects in the shared model. In this case the process of mapping to strings, or other fundamental objects, would be used.

CONCLUSION

The application views technique supports specification of the contents of user interfaces and reports. Views allow the many and varied users of a computer system, especially in MIS, to access common data in a way tailored to them. A distinction is made between views, which simplify information held in the complex shared model, and presentations, which format this information for the user. A view is defined through a series of methods that show how information in the view is obtained from and updates the shared model,

As stressed in the introduction, this approach is not a new O-O analysis and design method. Rather it is a technique that can be applied in conjunction with any of the O-O analysis and design methods currently published. It is a technique particularly suitable for MIS systems, and an adaptation of some venerable architectural principals. However the structure presented, it is, I hope, one that will aid anyone involved in the development of O-O systems. ■

Acknowledgments

The author thanks James Odell for his detailed comments on the manuscript. The ideas presented were developed when working on the Cosmos Project, which was supported by the Information Management Centre of the UK National Health Service. In this the author was helped by the other project team members: Tom Cairns, Mark Thursz, Anne Casey, Gerry Gold, Doug Simmonds, Michael Thick, and the lingering influence of Hazim Timimi.

References

1. Arnold, P. et al. An evaluation of five object-oriented development methods, JOOP FOCUS ON ANALYSIS & DESIGN, R.S. Wiener, ed., SIGS Publications, New York, 1991, pp. 101–122.
2. Monarchi, D.E. and G.I. Puhr. A research typology for object-oriented analysis and design, COMMUNICATIONS OF THE ACM, 35(9):35–47, 1992.
3. Kent, W. DATA AND REALITY, North-Holland, Amsterdam, 1978.
4. Griethuysen, J.J. INFORMATION PROCESSING SYSTEMS—CONCEPTS AND TERMINOLOGY FOR THE CONCEPTUAL SCHEMA AND THE INFORMATION BASE, Technical Report ISO/TR 9007:1987(E), International Organization for Standardization, 1987.
5. Jacobson, I. et al. OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Addison-Wesley, Wokingham, UK, 1992.
6. Martin, J. and J. Odell. OBJECT-ORIENTED ANALYSIS AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1992.
7. Rumbaugh, J. et al. OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1991.
8. Filman, R.E., W.S. Faught, and J. Solomon. The object-oriented development of a transaction-processing application, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, 5(7):51–60, 1992.
9. Wirfs-Brock, R., B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs NJ, 1990.
10. Gilb, T. PRINCIPLES OF SOFTWARE ENGINEERING MANAGEMENT, Addison-Wesley, Wokingham, UK, 1988.
11. Cairns, T. et al. THE COMSOS CLINICAL PROCESS MODEL, Report ECBS20A & ECBS20B, National Health Service, Information Management Centre, Birmingham, UK, 1992. (ftp from directory "cosmos" on dka.sm.ic.ac.uk)

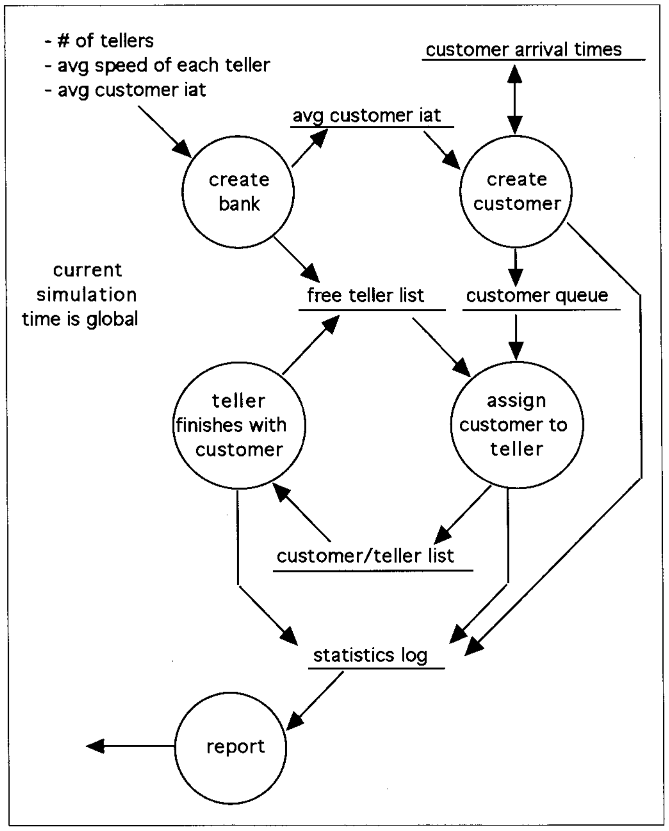

Figure 1. Dataflow for a procedural solution to the bank simulation

many new features, but the essential ones are objects, inheritance, and polymorphics.

An object encapsulates a group of procedures and some data. The code in the procedures references the data directly. The rest of the system passes parameters to the member procedures and does not reference the common data. The result is an abstract data type, or object. The payoff is that the implementation of the object can be changed without affecting the rest of the system. The classic example is a stack, with its push, pop, and clear functions, which can be implemented using either fixed or dynamic allocation. If you have a stack and want to change its implementation, the changes will be localized to push, pop, and clear, and the rest of the system will not have to be modified.

This is a good idea, but unfortunately *it doesn't solve the real maintenance problem.*

In program maintenance it is not normal to replace whole object implementations. For example, imagine that you had a program with several stacks, all built using fixed allocation. How likely is it that you would get a maintenance request to replace *all* of them with linked-list stacks? Not very likely. What is much more likely is that you would have to replace one of them or add another stack that is dynamically allocated.

The most common problem in maintenance is to add enhancements and custom modifications. In these cases we don't have to replace whole objects. Instead, what we usually have to do is *add new variants of existing objects.* When this is done, if the program is designed in terms of abstract data types alone any particular piece of calling code will only be able to reference one of these objects. Because the calling code has to know what kind of object is being called, the changes are not localized to the object. So abstract data types simply don't go very far in helping us solve the most common, most difficult problems of maintenance.

These problems are solved with polymorphism, generally linked with an object classification structure using inheritance. A single message can be interpreted differently by different objects, so we can write code that works regardless of variation in the objects on which it calls. We can tell a window to display itself without knowing what kind of window it is and, consequently, we can add new kinds of windows without having to add branching to the code that calls the display function. This solves some real maintenance problems, and it is *the distinguishing feature* of object-oriented design.

> The use of derived classes and virtual functions is often called object-oriented design. Furthermore, the ability to call a variety of functions using exactly the same interface—as is provided by virtual functions—is sometimes called polymorphism.[2]

Peter Wegner has proposed that we distinguish between *object-based* and *object-oriented* languages.[3] An object-based language (such as Ada) has abstract data types. An object-oriented language adds inheritance and polymorphic functions. This same distinction is useful in evaluating program designs. An *object-based design* is one that uses abstract data types. An *object-oriented design* is one that uses inheritance and polymorphism. To see why this distinction is important, we need to look at a design example. This example will show that abstract data types alone will not get you the benefits of object-oriented programming. The example given below is a simple program that simulates a bank:

> Customers enter the bank at randomly varying intervals, that is, the average customer inter-arrival time (iat) follows an exponential distribution. There are tellers, the number of whom can be set by the user when the program begins. When a customer enters, if a teller is available, the customer will go to the teller for service. When a teller serves a customer, the teller will be busy for a randomly varying amount of time. Each teller has a characteristic speed, which determines the mean around which the busy intervals will be distributed. If there are no tellers available when the customer enters, the customer goes into a queue. When a teller finishes with a customer (becomes unbusy), if there is a customer on the queue, the teller immediately serves the first customer on the queue.
>
> The program will run the bank for a day and collect summary statistics including the total number of customers served, the percentage of customers who had to enter the queue, the average wait time on the queue, the maximum length of the queue, the number of customers served by each teller, and the percentage of time busy for each teller.

This problem was used in an early book on C++[4] and likewise for Smalltalk.[5] In this article, we're going to look at three solutions to the problem—a functional solution, a naive "object-based" solution, and a true object-oriented solution. The goals of OOD will become clearer as we compare these designs.

Program 1: A functional solution

The first solution is a C program. Its data flow is shown in Figure 1.

The program begins with a create bank process that initializes vari-

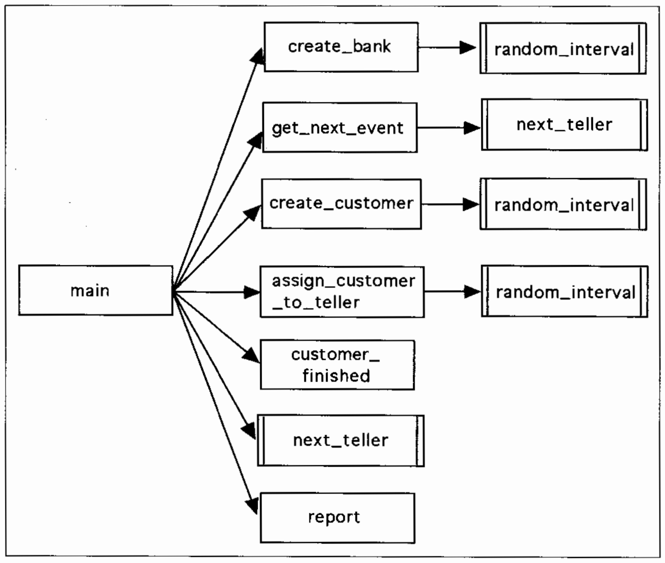

Figure 2. Main **module pseudocode and structure chart.**

Change	% of code affected
additional kinds of teller	76%
multiple queues	58%
customers can leave queue	41%
transaction types	87%
rush of customers	04%
tellers come and go	36%
diversion of resources	28%
cash machines	87%
average	52%

Table I. Effects of anticipated changes to specifications.

ables and allocates memory for tellers. New tellers are put on a free teller list. Once the variables are initialized, the main action is a circular flow of tellers. When a customer is served (assign customer to teller), the teller is moved from the free teller list to the (busy) customer/teller list. When a teller finishes with customer, the teller goes back to the free teller list. On the side, a create customer process generates new customers according to its own schedule. All these processes drop timing data into a log used to generate the reports.

The structure chart and main module pseudocode are shown in Figure 2. Each of the processes in the dataflow has become a module. A main module is put over top of everything to manage the flow of control, and several utility functions have been added.

```
Main Module
  create_bank
  while the bank is open
    if (the next_event is a customer_arrival)
      create_customer
    else if (next_event is teller_finishes_with_customer)
      let the customer_finish
    endif
    if (a teller_is_available && customerqueue is not empty)
      assign_customer_to_teller
    endif
  endwhile
  while tellers are busy
    let a customer_finish
    if customerqueue is not empty
      assign_customer_to_teller
    endif
  endwhile
  report summary statistics
```

So what's wrong with this design?

Recall that most of the effort in programming goes into maintenance, and most of maintenance involves program enhancement. To evaluate this design, put yourself in the place of the maintenance programmer. You will face an endless stream of requests for extension and custom modification. How can you take them into account?

Suppose we begin by making a list of *anticipated changes* to the program. Here is an example of such a list:

1. additional types of teller and other officers
2. use of multiple customer queues in various configurations
3. allowing for customers to leave the queue
4. incorporation of transaction types attached to customers
5. rush of customers at lunch and closing time
6. varying numbers of tellers, as when tellers go on break
7. ability of the bank to divert other resources to customer service when the line gets too long
8. cash machines and telephone transactions

This list will be our tool for criticizing the design. For example, suppose we have to make change #3—to allow customers to get impatient and leave the queue. To keep the queue length accurate at all times, it will be necessary to figure out when a customer should leave the queue, and actually remove him at that time. A function can be written to remove a customer from the queue, but it will be necessary to integrate this function into the program. main will have to be modified to call this function at the proper time. But in the original design, main determines the proper time to call a subfunction by collaborating with get_next_event. get_next_event compares the scheduled times of upcoming events, and tells main what should happen next. So both these functions will have to be changed. In addition, somebody will have to set the scheduled times for customer removal. The easiest way to do this is for create_customer to set these times into a table, which can be consulted by get_next_event. Finally, the report routine will change to show the effects of all this.

So four existing functions will have to be modified—main, get_next_event, create_customer, and report. The programmer who modifies these will have to understand (i.e., read) them. This gives us a metric for design quality. Because a good design should localize the effects of change, we can evaluate a design by getting a list of anticipated changes, and measuring how much of the existing code has to be understood to make each of the changes safely.

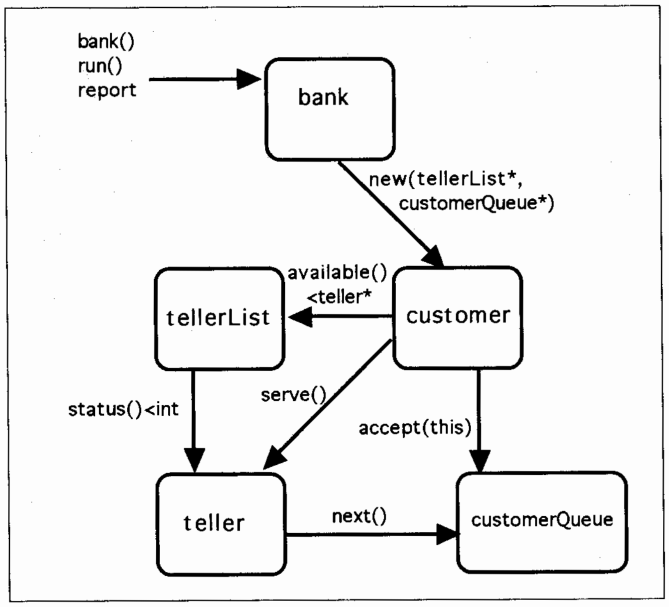

Figure 3. Coordination of tellers and customers.

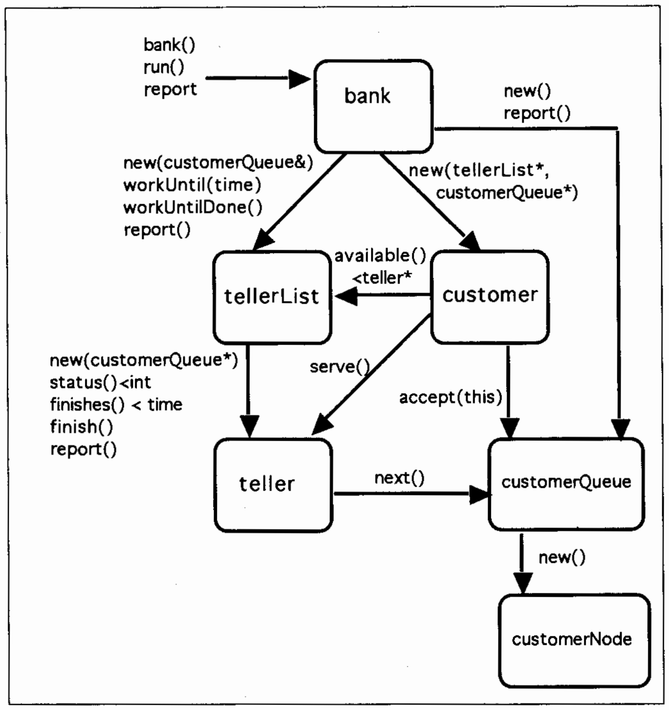
bank()
run()
report
bank
new() report()
new(customerQueue&) workUntil(time) workUntilDone() report()
new(tellerList*, customerQueue*)
tellerList
available() <teller*
customer
new(customerQueue*) status()<int finishes() < time finish() report()
serve()
accept(this)
teller
next()
customerQueue
new()
customerNode

Figure 4. A naive, object-based design.

In this case, the whole program is 113 lines of C code, of which main, get_next_event, create_customer, and report account for 46 lines, or 41%. So a maintenance programmer would have to study 41% of the existing program to extend it to allow customers to leave the queue.

We need some ground rules to guide this measurement. The rules I have used are:

1. A fixed amount of time should be allowed for figuring out how to make each change. The numbers I'm reporting result from spending 10 minutes designing each change. I actually spent 15 minutes, but I found that the extra 5 minutes had only marginal effect on the results. Use your judgment, but set a fixed time and be consistent in applying it.
2. If you're going to change any code in a function, you have to understand the whole function, i.e., functions must be counted as units. I did not count lines that contained only a brace.
3. If a change to a function affects its return value, you have to be familiar with all of the functions that depend on that return value.
4. If a change to a module writes into global data, you have to decide whether the meaning of the global data item has changed. If it has, then you will have to understand all of the other code that references that data.
5. If you're not sure whether a proposed change will affect a given function, assume that it will, because in either case you are probably going to have to study and understand that function.

The results of this analysis are shown in Table 1.

It is now quite clear what is wrong with this design. It does a dismal job of localizing the effects of change. On the average, about half the whole program is implicated in any given change.

Incidentally, the reason for this is that the program was designed using stepwise refinement. In other words, I began by writing the top-level module and testing it, stubbing off the lower-level modules. Thus the lower-level modules represent *steps in the execution* of the original program. Change the program, and you change the steps—they have no functional cohesion. The result is that the modularity of the program breaks down under maintenance. Stepwise refinement is a code-level design technique, not a system design technique. In fact, the initial paper on structured design warned against creating a module just because its "elements are executed in the same time period."[6] This was called "temporal" cohesion and was clearly identified as a poor criterion for modularization. But stepwise refinement is often taught in colleges, and system design is not, so this kind of program is actually quite common.

What we have done so far is to take an ordinary student program and examine it carefully, taking the point of view of the maintenance programmer. We measured the percentage of the total code that must be understood to make each of a number of enhancements. We found that the designer had not done a good job of localizing the effects of changes.

Now let's look at two solutions using objects.

Program 2: An object-based solution

To get the object-based solution I used a "naive" object-oriented design technique (a naive technique is one based on listing the domain objects). In my classes, I use Coad/Yourdon OOA for this.[7] For the last year I have been getting students to read the Coad/Yourdon book and then create a design for the bank simulation. Every single group has come up with five basic objects:

- customers
- tellers

- a queue for customers
- an array or list of tellers
- a bank to hold the whole thing together

Some groups add a "door" object that creates new customers, and there is a lot of variation in the treatment of simulation time. But for the core objects, everybody comes up with the same list.

There is less agreement about how to assign methods to classes. "Model the real world" breaks down at this point because the real world has different sequencing and efficiency constraints. In the design shown here, when a new customer is created, it asks the teller list if there is a teller available. The teller list polls its tellers for status. If the teller list finds a free teller, it returns that teller to the customer, who asks the teller for service. If there is no free teller, the customer puts itself on the queue. When a teller finishes with a customer, it asks the customer queue for the next customer (Fig. 3).

Scheduling is accomplished by cooperation between the bank and the teller list. The bank calls a random number routine to get the next customer arrival time, and then notifies the teller list that tellers can work until that time. Any teller who would finish before that time is then allowed to do so, by the teller list. The complete message diagram is shown in Figure 4.

Notice that this design does not use inheritance and polymorphics, so the naive OOD technique has produced an object-based rather than object-oriented design.

Once again we evaluate the design by estimating how much of the whole program has to be understood to make each of the anticipated changes. To deal with classes, there are two additional measurement rules:

6. To change a function in a class, you have to understand the whole function, the whole class declaration, and the declarations of parent classes if inheritance is involved.

7. If the change affects the value of any class variables, you have to decide whether the meaning of that variable has been changed. If it has, you will have to understand the other functions that access the variable.

Let's return to the example of allowing customers to get impatient and leave the queue. Because Customer and CustomerQueue are now encapsulated objects, they can manage this change. The Customer constructor can set a leaving time for each Customer. Whenever a Customer is added or removed from the CustomerQueue, CustomerQueue can poll the waiting Customers to see if any of them want to leave. So, this involves changing the Customer constructor, and most of the CustomerQueue class: 46 lines altogether, or 27% of the total code.

For the functional design, this change required modifying 41% of the code, so in this case the object-based design does better. Across the whole list of changes, however, the object-based design does not look so strong. Table 2 shows the side-by-side comparisons of the two designs. The reasoning for all the cases is given in the appendix.

As far as maintenance is concerned, the object-based design does not improve on the modularity of a sloppy C program. A financial officer who paid the cost of retooling to object-oriented methods would have good reason to be upset about results like these.

Change	% of program affected	
	naive OO design	functional design
additional kinds of teller	70	76
multiple customer queues	51	58
customers can leave the queue	27	41
transaction types	73	87
rush of customers	16	4
tellers can come and go	33	36
diversion of resources	38	28
cash machines	81	87
average	49	52

Table 2. Modularity metric for two designs

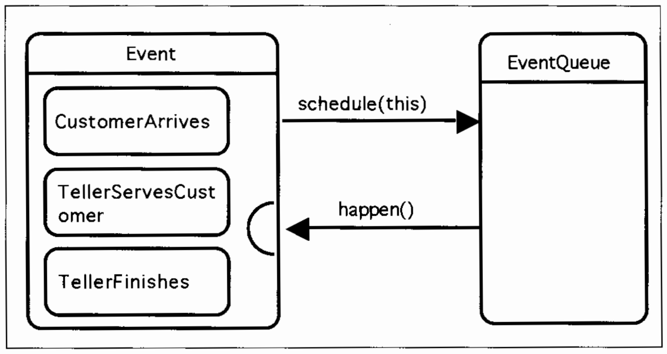

Figure 5: A polymorphic Event class, with scheduling handled by a sorted list. The happen() message is separately implemented in each of the three derived classes of Event.

I want to stress that I did not cook this example. Of course, it is possible to improve the way that methods are allocated to classes in the object-based design, but it is equally possible to improve the functional design. In each case, I merely fleshed out a "middle of the curve" student solution. It is the ordinariness of these designs that makes them interesting.

The most interesting thing about this problem is that if people are guided by the evident domain objects *everybody gets the same answer*, and *it's the wrong answer!* So let's look at a "right answer," and then ask how a design method might help us to find it.

Program 3: A true object-oriented design

The key to a better design for this problem lies in the complexity of the logic required to get the next event. The main problem in this program is figuring out *what kind of event* should happen next. Whenever a program is branching on "what kind of X," a polymorphic solution is indicated. This program needs an "event" class, with "customer arrives," "teller serves customer," and "teller finishes" as children, and polymorphic calls to replace the nested logic (Fig. 5).

CustomerArrives, TellerServesCustomer, and TellerFinishes all inherit from Event. Each Event will have a time when it is supposed to happen, and the virtual happen() method. Each class derived from Event must implement happen() and also provide a constructor.

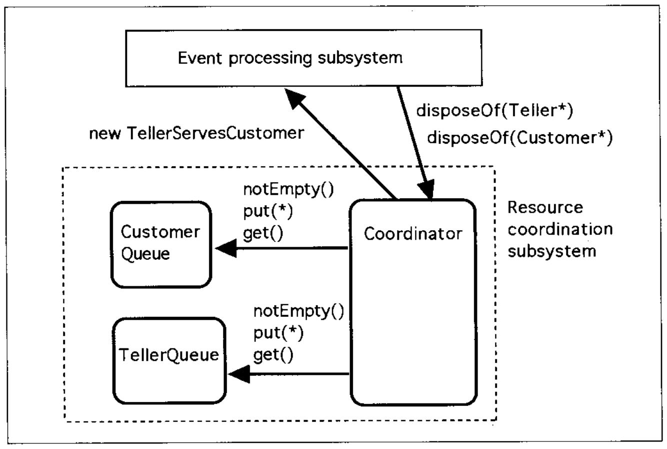

Figure 6. The resource coordination subsystem.

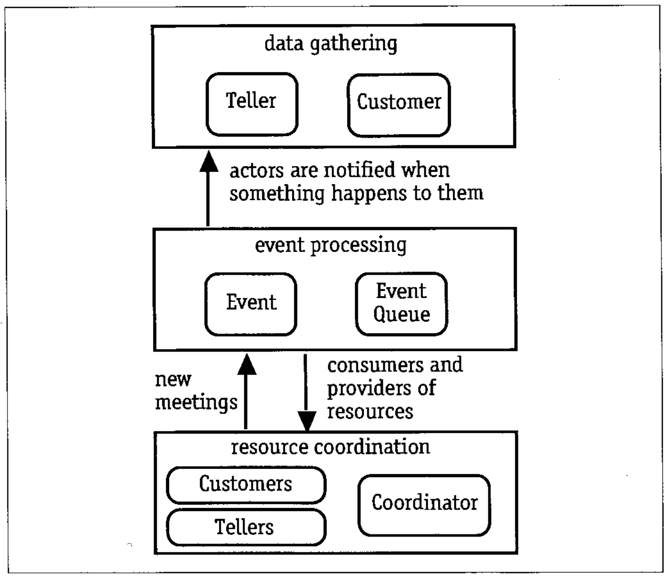

Figure 7. Subsystem diagram—a true object-oriented design.

When the Event is created, its constructor determines its "happening time," and sends a schedule message to EventQueue, a priority queue that sorts the Events according to time. That is, events can be put on the queue in any order but a next() message sent to the queue will cause the event with the earliest time to happen. So, the responsibilities of the Event classes are to decide when they should happen and carry out appropriate processing ("happen") when that time comes. The responsibilities of the EventQueue are to sort events according to time and cause the next event to happen. The main loop of the program creates the first customer arrival, and then begins popping events off the queue and letting them happen:

```
new CustomerArrives;
while (eventQ.notEmpty()) eventQ.next();
```

Of course, what happens on each pass of this loop depends on what kind of event is at the top of the queue. A CustomerArrives event will create a customer and send it off to be served by a teller or queued. A TellerServesCustomer event will notify the teller and create a CustomerFinishes event. The CustomerFinishes event will notify and free the teller.

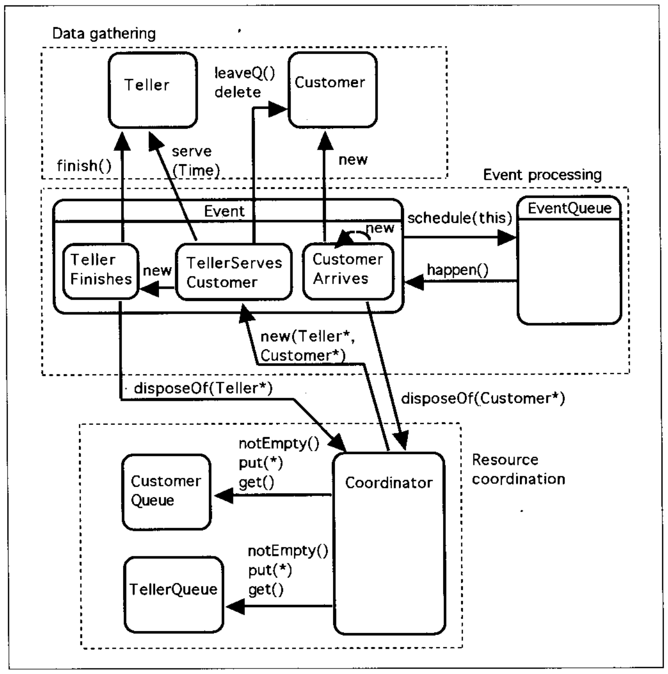

Figure 8. Message diagram.

In designing these event processing classes, we have built a *subsystem.* A subsystem is a group of classes that work together to provide a portion of the overall system functionality. There are two additional subsystems. The first is called the *resource coordination* subsystem: it matches customers with tellers. It has a queue for tellers, a queue for customers, and a *coordinator.* Newly arrived customers and available tellers are sent to the coordinator. When the coordinator matches a teller and a customer, it creates a new TellerServesCustomer event. This event will schedule itself to happen immediately. When it happens, it will set the teller busy and create a corresponding TellerFinishes event (Fig. 6).

The last subsystem is responsible for keeping statistics and reporting. This "data-gathering" subsystem contains the Customer and Teller classes. Their only job is to keep track of when things happen to them. For example, when a TellerServesCustomer event causes a teller to become busy, it will notify the teller, which will perform the recordkeeping. Notice that this is a dramatic change in the meaning of the Customer and Teller classes. Figure 7 is the complete subsystem diagram.

This program makes use of library classes for the lists and queues. In particular, EventQueue is a sorted list. The sorting logic inherited from the library replaces all the scheduling logic in previous versions of the program. We also don't have to worry about what kind of event is happening, because that is handled by the polymorphic call. The result is a dramatic drop in program complexity—in the whole program there is one loop and three branches!

The complete message diagram is shown in Figure 8. The code is shown and some of the initial processing is explained in the appendix.

Now we turn to the effects of the anticipated changes on this program. For example, to allow customers to leave the queue, a new type of Event called CustomerLeavesQueue will have to be written. The Customer class will be modified to interact with these events, as follows. The Customer constructor will create a CustomerLeavesQueue event for each Customer. That event will schedule itself. When a Customer is served, it will notify its associated CustomerLeavesQueue event so the event can inactivate itself. The Customer::report function will have to be changed. To make these changes, it is necessary to understand the Event class (10 lines) and the Customer class (20 lines), for a total of 20% of the whole program. Table 3 compares all three designs for modularity with respect to the anticipated changes.

The quality of the new design is striking. On the average, a change to this design impacts half as much code as with the other two designs.

The key to this design is the Event class. Once this class is discovered, the rest of the design is fairly straightforward. The class works for two reasons. The first is that the polymorphic happen function reduces the complexity of the whole program by replacing program logic. The second reason is more important—most anticipated changes to this program involve adding new kinds of events to the scheduling loop, and the Event class "objectifies" those changes.

Now this class is simply not in the original specification, which talks about "objects" like Customers and Tellers. The final design has Customers and Tellers, but they no longer run the action, and they are not modeled on the domain objects after which they are named.

In addition to adding new kinds of events to the scheduling loop, many of the anticipated changes affect the ways that customers enter queues and get matched with tellers (e.g., different teller types, adding separate queues for different tellers). In this design, these changes are encapsulated in the Coordinator class. Again, this class is not in the original specification, so its existence runs directly counter to the notion that we should model the domain objects. But this class results in a significant improvement in the maintenance situation.

DISCUSSION

What does this tell us about object-oriented design?

First, it tells us that there is more to identifying classes than just listing the domain objects. People who design in terms of objects "there for the taking" will produce object-based designs. The use of polymorphic functions will not be central to these designs and it will not be creative. The best we can expect is that an obvious classification will be noticed, as when a program dealing with cars has subclasses for Fords and Chevys.

On the other hand, when we look at clever and creative designs, they always seem to contain classes not quite evident in the domain. We have looked at one example in great detail. Another example is the "Command" class used in some application frameworks.[8] The Command class specifies an interface with two (virtual) methods—"do" and "undo." A subclass of Command might be "Delete_line." When the user invokes the command, an instance of the appropriate Command object is created, and the "do" message is sent to it. It stores the information it would need to undo itself. While this class seems natural after we've seen it, it is not clear how we can teach people to come up with such a class, starting from the specification for a text editor. The "objects" in the domain of text editing are characters, lines, and so on, and do not obviously include commands. Yet this class is the key to building a truly extensible system.

Change	true object oriented	object based	functional
additional types of teller	32	70	76
multiple customer queues	23	51	58
customers can leave the queue	20	27	41
transaction types	49	73	87
rush at lunch and closing	9	16	4
varying numbers of tellers	32	33	36
diversion of resources	18	38	28
cash machines	41	81	87
average	28	49	52

Table 3. Percentage of total code that must be understood to implement changes to three designs.

I want to call these the "great classes." Such classes simplify and organize large parts of a program. There are many ways to find these classes including:

1. be a genius
2. have lots of time
3. solve the problem three times
4. know the pattern (architecture) from previous experience or study
5. have a method that helps you get there.

These are all good ways to do it, but what we are talking about right now is #5—a *method* that will help us to find classes beyond our current experience.

How can we find the great classes? One approach is to have a list of heuristics to help us think better. For example, a "Command" is a kind of an Event object. Because this kind of object is easy to overlook, all the object-oriented design books have a "what to look for" list that includes "events." I have never seen these lists help anybody. In fact, I have emphasized these lists to my students, and still none of them ever come up with anything like the Event class in the bank example. It is not enough to tack notes onto a method that doesn't quite work. We need a more general solution.

The great classes are essentially interface specifications. If we are writing in C++, this interface will be embodied in an abstract class. We call it abstract because to find it we have to abstract from the concrete objects, i.e., group them and forget about variation in the group. How do we group them and what do we forget about? Now we've come to the crux of the problem. *The only way to do this abstraction is to make the list of anticipated changes, because those are precisely the variations from which the abstract class should protect us.*

This grouping and forgetting is not a combinatorial process on existing objects. We didn't have a bunch of individual Event objects lying around, with attributes in common—we recognized the abstract class and the concrete classes simultaneously. Further, we don't create the abstract class because it is evident in the domain—we create it because there are changes coming in the domain and we want to control their impact on the system. The great classes are based on change rather than permanence, on what we do not want to see rather than what we see. Objects like this are not found in the evident structure of the domain—they can only be found by describing and analyzing what we expect to happen to the program over time. Rather than ask, "What do you see there?" we should ask "What kinds of variation would you like to be able to ignore?" In the bank simulation, most of the anticipated changes involve (1) adding new kinds of events and (2) controlling the match between a teller and a customer. These two kinds of problems are solved in the Event and Coordinator classes.

I have been calling this "defensive analysis." Ivar Jacobson and his group at Objective Systems speak of "robustness analysis," which has similar goals although the process is different. Erich Gamma has said that the designer must "objectify variations," i.e., identify a kind of variation and create an object to contain it.[9]

It is not so hard to do this. Programmers experienced in a job can easily come up with a list of expected modifications and extensions to the programs on which they are working. If you can specify a program, you can also specify a list of anticipated changes. The next step is to abstract and generalize from this list. If you think about what must be done to implement each of the changes, you will discover recurring problems. Adding an enhancement or modification to the program often involves finding a new solution to one of these problems. When you can identify such a problem, specify the interface for a class or subsystem that will solve it. These objects then must be merged with the objects found from more straightforward domain analysis.

One common charge against object-oriented programmers is that they like to hack their way to a solution rather than use a design method. Object-oriented programming grew up without an associated analysis or design. The common explanation for this is that "Smalltalk is so powerful, they don't need a method." I think the reason is quite different. Object-oriented programmers have extensibility and flexibility as goals. To get these, you need to have experience with change. Thus, *criticizing an existing design is an integral part of object-oriented design.* You need a list of required changes to a design to find the classes that will allow you to make changes easily. This leads experienced object-oriented programmers to feel most comfortable in a prototype-driven, evolutionary development cycle.

What I am proposing in this article is that we can jumpstart this critical process by trying to anticipate the changes that the program will face. This is either the last round of design or the first round of maintenance. In either case, it is a way to build a bridge from design to maintenance. One argument in favor of object-oriented programming is that it provides a framework for a unified software method. Naive, object-based methods unify analysis, design, and coding. True object-oriented methods unify these with maintenance.

SUMMARY

Discussions of the benefits of object-oriented programming often emphasize reuse rather than extensible, maintainable systems. Reuse applies to individual classes and pays off several years down the road. Maintainability applies to whole systems and pays off sooner. So all the talk about reuse is probably a marketing mistake, and certainly makes it harder to think clearly about system design. Because we are really concerned with building systems, maintainability is a more important goal than reuse.

If we focus on maintainability, we are rewarded with clear guidelines for system design. A maintainable object-oriented design is built around a central family of classes that replace switching logic with polymorphic calls. These classes are characterized by the variation they hide rather than what they are.

Such classes will not be found by listing the evident domain objects. This implies that many popular OOD methods cannot be expected to produce truly object-oriented designs. To build systems that are robust under change, it is necessary to anticipate and abstract from expected changes. If we sell methods that don't do this, we are going to have a lot of disappointed customers in a few years time.

Whatever method or notation you use, two additional activities need to be added to analysis and design. The first is a "requirements generalization" activity, in which a list of expected changes is compiled. The second is a critical examination of the initial design in terms of its robustness to these changes. This step involves "objectifying the variations" that have been listed. It can be thought of as a first pass at program maintenance. Percent of the total code affected by a change can be used as a metric for comparing designs.

Object identification heuristics should emphasize the information-hiding aspect of objects. It's all right to ask "What's in front of my nose?" But you should also ask questions such as the following as guides for identifying objects:

- "What is the most difficult problem in resolving this design?"
- "What kinds of changes will have to be made to this program during maintenance?"
- "How will these changes be made?"
- "Are there significant, recurring problems in making these changes?"
- "What is the one problem that I wish were already solved?"
- "If I had a class that solved this problem, what would it do for me?"
- "What is true about this class, no matter how you solve the problem?"

These questions will help designers identify abstract classes. Of course, an abstract class does not "solve" a problem, it merely establishes a protocol by which solutions can be requested and reported.

In fact, the maintainability goal does not conflict with reuse, if the two are taken in the right order. The significant cases of reuse involve frameworks rather than single components—so-called *design reuse.* And frameworks like MVC, ET++, and MacApp use polymorphism as described in this article. In other words, *the first step towards reuse is to design for maintainability.*

The role of maintenance in design, as well as the proper goals of design, are described by Winograd and Flores:

> . . . to anticipate the forms of breakdown and provide a space of possibilities for action when they occur. It is impossible to completely avoid breakdowns by means of design. What can be designed are aids for those who live in a particular domain of breakdowns.[10] ■

References

1. Jacobson, I. et al. OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Addison-Wesley, Reading, MA, 1992.
2. Ellis, M.A. and B. Stroustrup. THE ANNOTATED C++ REFERENCE MANUAL, Addison-Wesley, Reading, MA, 1990.
3. Wegner, P. Concepts and paradigms of object-oriented computing, OOPS MESSENGER 1(1), 1990.
4. Wiener, R.S. and L.J. Pinson. AN INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING AND C++, Addison-Wesley, Reading, MA, 1988.
5. Goldberg, A. and D. Robson, SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION, Addison-Wesley, Reading, MA, 1983.
6. Stevens, W., G. Myers, and L. Constantine, Structured design, IBM SYSTEMS JOURNAL, 1974, reprinted in CLASSICS IN SOFTWARE ENGINEERING, E.N. Yourdon, Ed., Yourdon Press, New York, 1979.
7. Coad, P. and E. Yourdon. OBJECT-ORIENTED ANALYSIS, 2ND ED., Yourdon Press, New York, 1991.
8. Meyer, B. OBJECT-ORIENTED SOFTWARE CONSTRUCTION, Prentice Hall, Englewood Cliffs, NJ, 1988.
9. Gamma, E. comments during a question and answer session following presentation of the paper, ET++ SwapsManager: Using object technology in the financial engineering domain, OOPSLA '92.
10. Winograd, T. and F. Flores, UNDERSTANDING COMPUTERS AND COGNITION, Ablex, 1986.

APPENDIX

For the object-based and object-oriented solutions, this appendix contains the proposed strategy for making each of the eight changes. For the object-oriented solution, it includes a description of some of the initial event processing and a code listing. The details for the functional solution are available from the author.

Object-based solution

Proposed approaches for making each of the anticipated changes, showing the modules affected and their lengths:

1. additional kinds of teller
   - TellerList will have to be elaborated into a teller list manager (45)
   - the Customer will have to know what kind of Teller it wants. This means a change to the Customer constructor (8 + 9)
   - Customer will send some information about the kind of officer wanted in the call to TellerList::available (10 + 4)
   - the Teller's call to CustomerQueue::next will have to make a similar coordination (16 + 4 + 9 + 10)
   - total: 115 lines
2. multiple customer queues
   - the TellerList will have to associate a CustomerQueue with a group of Tellers (45)
   - the Bank cannot pass CustomerQueue directly to the Customer for initialization—the Customer will have to get that from the TellerList—this changes Bank::run and the Customer constructor—(10 + 8 + 8 + 9)
   - the Bank will not be able to ask the CustomerQueue to report—that also will have to be passed through TellerList (4)
   - total: 84 lines
3. customers can leave the queue
   - customers will get an "impatience rating" when they are created (8 + 9)
   - Whenever a Customer is added or removed from the CustomerQueue, it will poll all the Customers on the queue to see if any of them want to leave (29)
   - total 46 lines
4. transaction types associated with customers
   - probably involves additional Teller types (115)
   - the Customer constructor will have to determine the transaction type for that Customer
   - Teller::serve will have to change (4)
   - total: 119 lines
5. rush of customers at lunch and closing time
   - the Bank constructor and Bank::run are the only places that Customers are created (10 + 8 + 8)
   - total: 26 lines
6. varying numbers of tellers
   - this can all be handled by the TellerList (the whole class has to be checked)
   - Teller::report will also have to change (13 + 3)
   - total: 61 lines
7. diversion of other resources to customer service
   - this can be handled by the TellerList, but it will be necessary to get the current length of the CustomerQueue (45 + 17)
   - total: 62 lines
8. cash machines
   - Combines additional types of Tellers, customer Transaction types, and multiple CustomerQueues. Between these, they affect every class in the program except Teller
   - total: 133 lines

Object-oriented solution

The code follows. When the program begins, the event queue is empty. main schedules the first CustomerArrives event. Because it is the only event on the queue, it will be the first event to happen.

continued

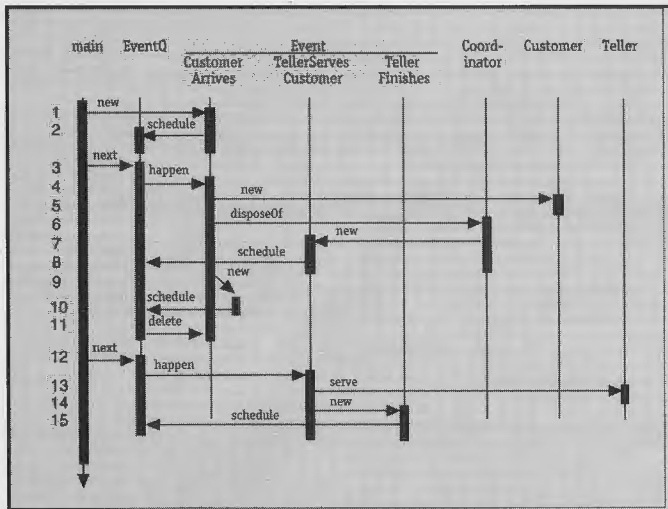

Figure 9. The first two events happen.

When the first customer is created, the coordinator will immediately match that customer with a teller because all the tellers are initially free. Each CustomerArrives event, when it happens, creates its successor, the next CustomerArrives. Thus, the initial processing is:

1. main creates the first CustomerArrives,
2. which schedules itself. The event queue now holds one event.
3. main asks the event queue to perform the next (i.e. the first) event.
4. The event queue gets the CustomerArrives event, and tells it to happen.
5. The CustomerArrives event creates a Customer,
6. and sends it to the Coordinator.
7. The Coordinator matches the Customer with a Teller, and creates a TellerServesCustomer event,
8. which schedules itself to happen immediately.
9. The CustomerArrives event creates the next CustomerArrives event,
10. which schedules itself.
11. The event queue deletes the first CustomerArrives, which has just happened.
12. main asks the event queue to perform the next event.
13. The TellerServesCustomer event notifies the Teller to get busy,
14. and creates a TellerFinishes event
15. which schedules itself.

This sequence is shown in Figure 9 using an interaction diagram.[1] When it is finished, four events have been scheduled, and two have happened. There are two events waiting on the event queue —a TellerFinishes (because one teller is busy) and a CustomerArrives (because as long as the bank is open, there is always one customer arrival scheduled).

Interactions between the Coordinator and its queues are not shown.

Here is a proposed strategy for handling each of the expected changes:

1. additional types of tellers and other officers
   - Coordinator will have to keep more queues (18)
   - Tellers will have to remember their types and this should be included in the report (12 + 5)
   - The Customer constructor will have to be modified to determine what kind of Teller type the Customer will need and to remember this information. The Coordinator will match the appropriate kinds of Customers and Tellers when creating the TellerServesCustomer events (13)
   - total: 48 lines
2. use of multiple customer queues
   - The Coordinator will be completely changed (18)
   - Tellers can remember to which CustomerQueue they are attached (12 + 5)
   - total: 35 lines
3. allowing customers to leave the queue
   - The new Event subclass (CustomerLeavesQ)will have to be written (10)
   - The Customer class will be modified to interact with these new events as follows: the Customer constructor will create one of them, which will schedule itself. Each Customer will keep a reference to its CustomerLeavesQ event. When a Customer is served, it will notify its own queue leaving event, which will inactivate itself. The Customer::report function will have to be changed to report these events (20)
   - total: 30 lines
4. transaction types and multiple transactions
   - If Customers are going to have transaction types, it is likely that Tellers will be specialized, so this change is equivalent to the first one—additional types of Tellers, except that in this case the existing TellerServesCustomer class will probably not do any longer—(48 + 16 + 10)
   - total: 74 lines
5. rush of customers at lunch and closing time
   - The CustomerArrives class will be modified
   - total: 14 lines
6. varying numbers of tellers
   - new child classes of Event will be added and must be scheduled from main (10 + 22)
   - the Teller report routine will change (12 + 5)

- total: 49 lines

7. diversion of resources to customer service
   - the Coordinator will have to change so as to measure queue lengths when customers are inserted and schedule diversion events when necessary. The Queue classes that the Coordina tor uses come from a library and are already capable of giv ing their lengths (18)
   - new Event class to add Tellers (10)
   - total: 28 lines
8. cash machines
   - the Customer constructor has to determine the customer's needs (13)
   - the Coordinator will have to be extended (18)
   - there will have to be a new type of Teller object (22)
   - there will have to be a new kind of Event object (10)
   - total: 63 lines

Program #3: Object-oriented solution

```
#include <math.h>

// templates and libraries (available from the author)
#include "queues.tem"
#include "pqueue.hpp"
#include "time1.hpp"

#define NL << "\n"

class Customer;
class Teller;
class Event;
class CustomerArrives;
class TellerServesCustomer;
class TellerFinishes;
class EventQueue;

Time currentTime;
Time startTime;
Time closingTime;
//
**********************************************
// The Coordinator class is written using templates, to make it useful
// outside the context of Customers and Tellers. It matches a Consumer
// with a Server to produce a Meeting.

template <class Consumer, class Server, class Meeting>
class Coordinator {
    Queue<Consumer> cq;
    Queue<Server> sq;
public:
    void disposeOf(Consumer*);
    void disposeOf(Server*);
    Queue<Consumer> *getCQ() {return &cq;}
    Queue<Server> *getTQ() {return &sq;}
};
template <class Consumer, class Server, class Meeting>
void Coordinator<Consumer, Server, Meeting>::disposeOf(Consumer *c) {
    if (sq.notEmpty())
            new Meeting(sq.get(), c);
    else
            cq.put(c);
}

template <class Consumer, class Server, class Meeting>
void Coordinator<Consumer, Server, Meeting>::disposeOf(Server *t) {
    if (cq.notEmpty())
            new Meeting(t, cq.get());
    else
            sq.put(t);
}
// end of Coordinator template
//
***********************************

class Customer {
    Time arrivalTime;
    static int totalCustomers;
    static Duration timeInQueue;
public:
    static void initializeStatistics() {
            totalCustomers = 0;
            timeInQueue = 0;
    }
    Customer() {arrivalTime = currentTime;}
    void leaveQueue() {
            timeInQueue += (currentTime - arrivalTime);
            totalCustomers++;
    }
    static void report();
};

class Teller {
    Duration averageServiceTime;
    int customersServed;
    Duration timeInService;
public:
    Teller();
    Duration serve() {
            Duration timeBusy = averageServiceTime.randomInterval();
            timeInService += timeBusy;
            customersServed++;
            return timeBusy;
    }
    void finish() {}
    void report();
};

class Event {
protected:
    Time happeningTime;
    static EventQueue *eq;
    static Coordinator<Customer,Teller,TellerServesCustomer>*coord;
public:
    static void setQueues(EventQueue *e,
            Coordinator<Customer,Teller,TellerServesCustomer> *c)
    {
            eq = e; coord = c;
    }
    virtual void happen() = 0;
    Time getTime() {return happeningTime;}
};

class CustomerArrives: public Event {
    static Duration avgInterArrivalTime;
public:
    static void initializeIAT(Duration d) {
            avgInterArrivalTime = d;
    }
    CustomerArrives();
    virtual void happen();
};

class TellerServesCustomer: public Event {
    Teller *theTeller;
    Customer *cust;
public:
    TellerServesCustomer(Teller*, Customer*);
    void happen();
};

class TellerFinishes: public Event {
    Teller *theTeller;
public:
    TellerFinishes(Teller*, Time);
    virtual void happen();
};
int eventCompare(void *e1, void *e2) {
    Time t1 = ((Event*)e1)->getTime();
    Time t2 = ((Event*)e2)->getTime();
    Duration t3 = t1 - t2;
    if (t3 > .001) return 1;
    else if (t3 < -.001) return -1;
    else return 0;
}

class EventQueue: private pqueue {
public:
    EventQueue(): pqueue(eventCompare) { }
    void next() {
            Event *e = (Event*)pqueue::get();
            e->happen();
            delete e;
    }
    void schedule(Event *e) {pqueue::put(e);}
    int notEmpty() {return pqueue::notEmpty();}
};

void Customer::report() {
    cout << "Current time is: " << currentTime NL
            << "While the bank was open, " << totalCustomers
            << " were served." NL
            << "They spent an average of "
            << (timeInQueue/totalCustomers)
            << " minutes waiting in queue." NL;
}

Teller::Teller() {
    cout << "Enter a teller's average service time in minutes: ";
    cin >> averageServiceTime;
    customersServed = 0;
    timeInService = 0.0;
}

void Teller::report() {
    cout << "                              " << customersServed
            << "                              "
            << timeInService/(currentTime - startTime) * 100
            << "\n";
}

CustomerArrives::CustomerArrives() {
    happeningTime=currentTime+avgInterArrivalTime.randomInterval();
    if (happeningTime < closingTime) eq->schedule(this);
    else delete this;
}

void CustomerArrives::happen() {

    currentTime = happeningTime;
    coord->disposeOf(new Customer);
    new CustomerArrives;
}

TellerServesCustomer::TellerServesCustomer(Teller *tp, Customer *cp) {
    theTeller = tp; cust = cp;
    happeningTime = currentTime;
    eq->schedule(this);
}

void TellerServesCustomer::happen() {
    cust->leaveQueue();
    delete cust;
    Duration busy =           theTeller->serve();
    new TellerFinishes(theTeller, currentTime + busy);
}

TellerFinishes::TellerFinishes(Teller *tt, Time tm) {
    theTeller = tt;
    happeningTime = tm;
    eq->schedule(this);
}

void TellerFinishes::happen() {
    currentTime = happeningTime;
    theTeller->finish();
    coord->disposeOf(theTeller);
}

Duration CustomerArrives::avgInterArrivalTime;
int Customer::totalCustomers;
Duration Customer::timeInQueue;
EventQueue *Event::eq;
Coordinator<Customer,Teller,TellerServesCustomer>* Event::coord;

void main() {
    // Initializations
    EventQueue eq;
    Coordinator<Customer, Teller, TellerServesCustomer> coord;
    Event::setQueues(&eq, &coord);
    Customer::initializeStatistics();
    cout << "Enter the time the bank will remain open\n"
            << "hours minutes, blank separated:\n";
    cin >> closingTime;
    cout << "Enter the average time between customer arrivals\n";
    Duration d;
    cin >> d;
    CustomerArrives::initializeIAT(d);
    Queue<Teller> *tq = coord.getTQ();
    cout << "How many tellers will the bank have? ";
    int nt;
    cin >> nt;
    for (int i = 0; i < nt; i++)
            tq->put(new Teller);

    // Main loop
    new CustomerArrives;
    while (eq.notEmpty()) eq.next();

    // Reporting
    Customer::report();
    cout << "\n\nTeller activity:\n"
            << "                              Customers Served"
            "                 % of time busy\n";
    while (tq->notEmpty()) tq->get()->report();
}
```