Family traits in business objects and their applications

by R. Prins
A. Blokdijk
N. E. van Oosterom

The business information system of the future will take the form of a swarm of business objects that are event-driven, concurrently executing, and running in a heterogeneous distributed environment. The inherent complexity of the business-object development process requires a difficult-to-find combination of skills in its developers. This complexity needs to be reduced to enable the participation of typical developers and to yield more successful projects. Fortunately, there are many common aspects among business objects. This paper describes a development approach that exploits these commonalities, reducing complexity through systematically defined, separate layers. The approach was developed in a research effort performed by the Application Development Effectiveness practice of the IBM Consulting Group in the Netherlands. It was subsidized by the Dutch Ministry of Economic Affairs as an information technology innovation project. A "proof of concept" was obtained in a joint project with Rabobank in the Netherlands. The result is a component-based development process with well-defined reuse points and rapid-application-development (RAD) characteristics. With this approach, robust business objects can be developed and tested individually and concurrently in large teams, then dynamically assembled into business applications and workflows as desired.

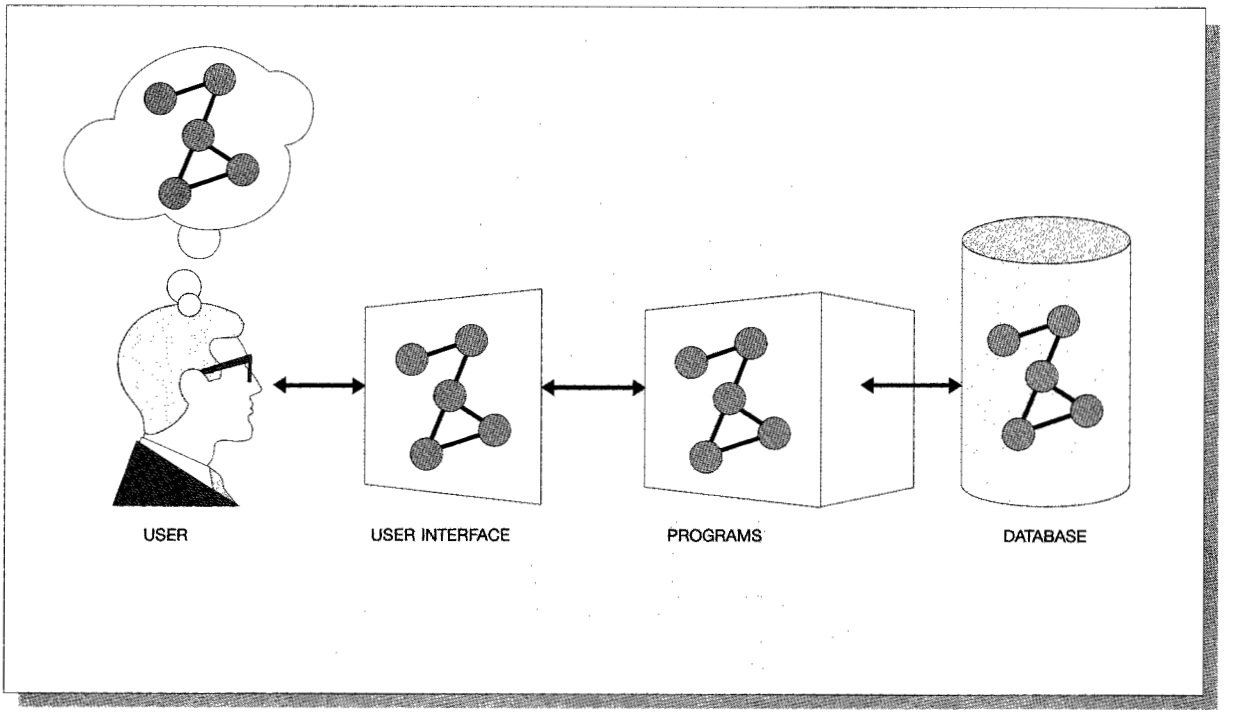
In this paper we summarize the results of our research on developing an efficient method for producing business objects. First we describe the business-object concept and the potential for software reuse that it provides. Next we describe the production method that we developed through our research. The method description is in two parts: analysis and synthesis. The analysis section describes how the elements of the business objects are determined. The synthesis section describes how the elements resulting from analysis are assembled together into business objects. We were able to achieve a high degree of uniformity across different problem domains in the synthesis of business objects. We now have a complete method to design and code robust business objects at prototyping speed. It is a practical method based on our research with customer projects. In our final section we discuss some of the ways that these customer projects influenced our research results.

What business objects have in common

The purpose of a business is to deliver products to a market in order to make a profit. In general, a business has a number of processes, each of which can be characterized as a pattern of business events. A business also has a domain that can be characterized as a collection of business objects. Business objects are the "things" around which each business process is organized. Examples of business objects are products, customers, units in which the customers consume the products and in which the business produces them (seen in customer orders), resources that are contributed during the production process

0018-8670/97/$5.00 © 1997 IBM

Figure 1 An object-oriented business information system mirrors reality. Its model includes the business objects that it maintains and provides information about to the user, and it closely resembles the user's conceptual model of the business.



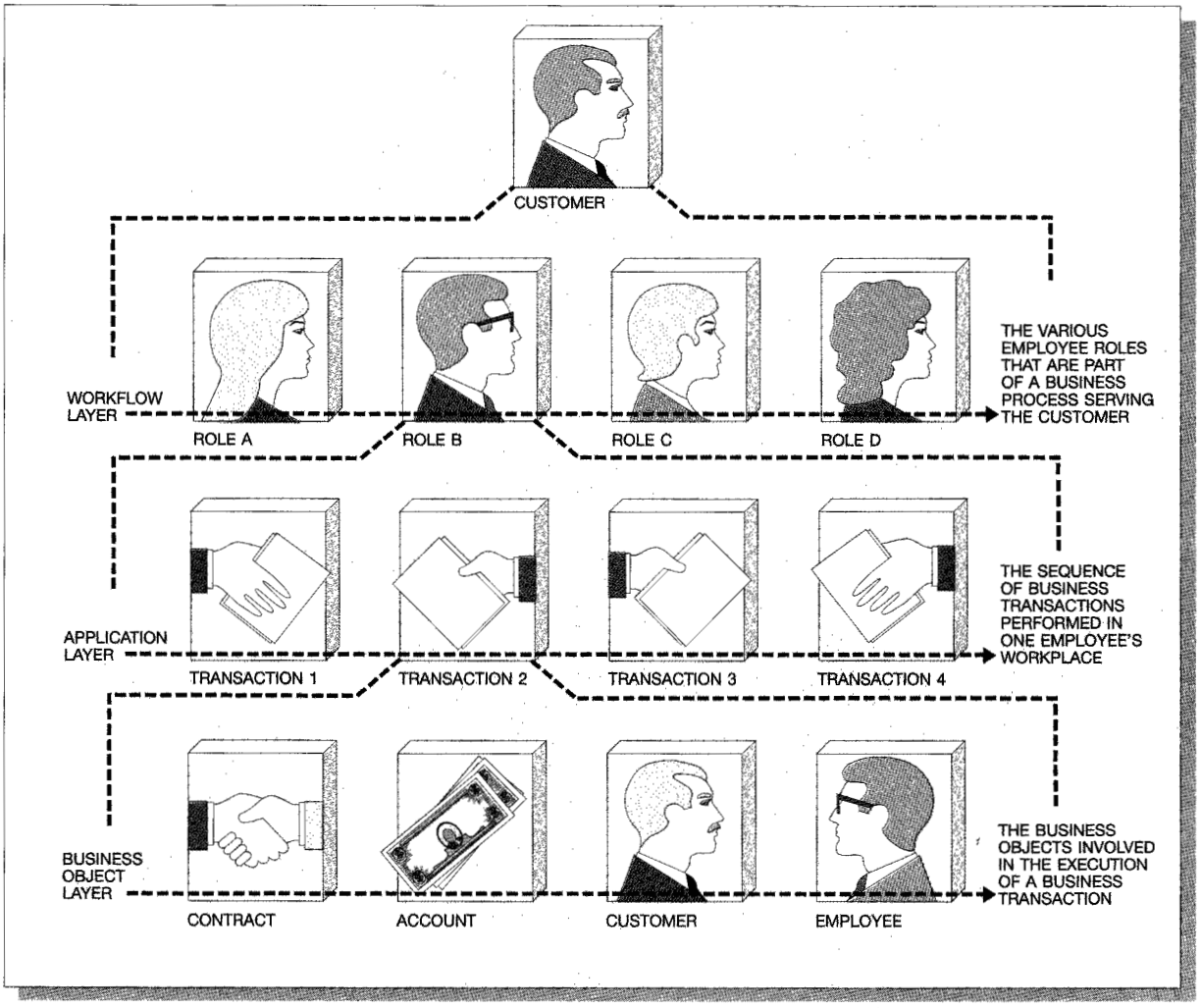
(materials, employees, machines), units in which these resources are acquired (seen in purchase orders), and suppliers of these resources.

A business needs to maintain current information about the business objects in its domain. A business information system must reflect the business objects that it maintains and uses to provide information. The points where information is requested or changed are found in the business processes. The business objects are not self-propelled. At every point in their life cycle they are dependent on externally supplied triggers. For this they need to be connected to business processes. So the business information system must be equally well adapted to both the business objects and the business processes. The business information system does not lead, it follows. It models the static and the dynamic aspects of its business world as closely as possible (Figure 1).

Modeling the business objects and the business processes yields the functional requirements of an information system. In addition, there are many nonfunctional requirements. They specify the infrastructure on which the information system must run. Examples are the server platforms, client platforms, network type, database mechanism, presentation form, national languages supported, security provisions, international standards, etc. In systems analysis the emphasis is on the functional requirements, but in design and implementation most of the work is on the nonfunctional requirements. Building a business object is like creating an iceberg. As it floats around in the company information system, all that is visible is the 10 percent that implements the functional requirements. Most of it is hidden from sight; 90 percent of the volume is needed to meet the nonfunctional requirements.

For a given business information system, the nonfunctional requirements are constant across otherwise very different business objects, e.g., product, customer, and account. This commonality allowed us to develop a generic business object that met a large number of our nonfunctional requirements. Later we specialized the generic object, applying the func-

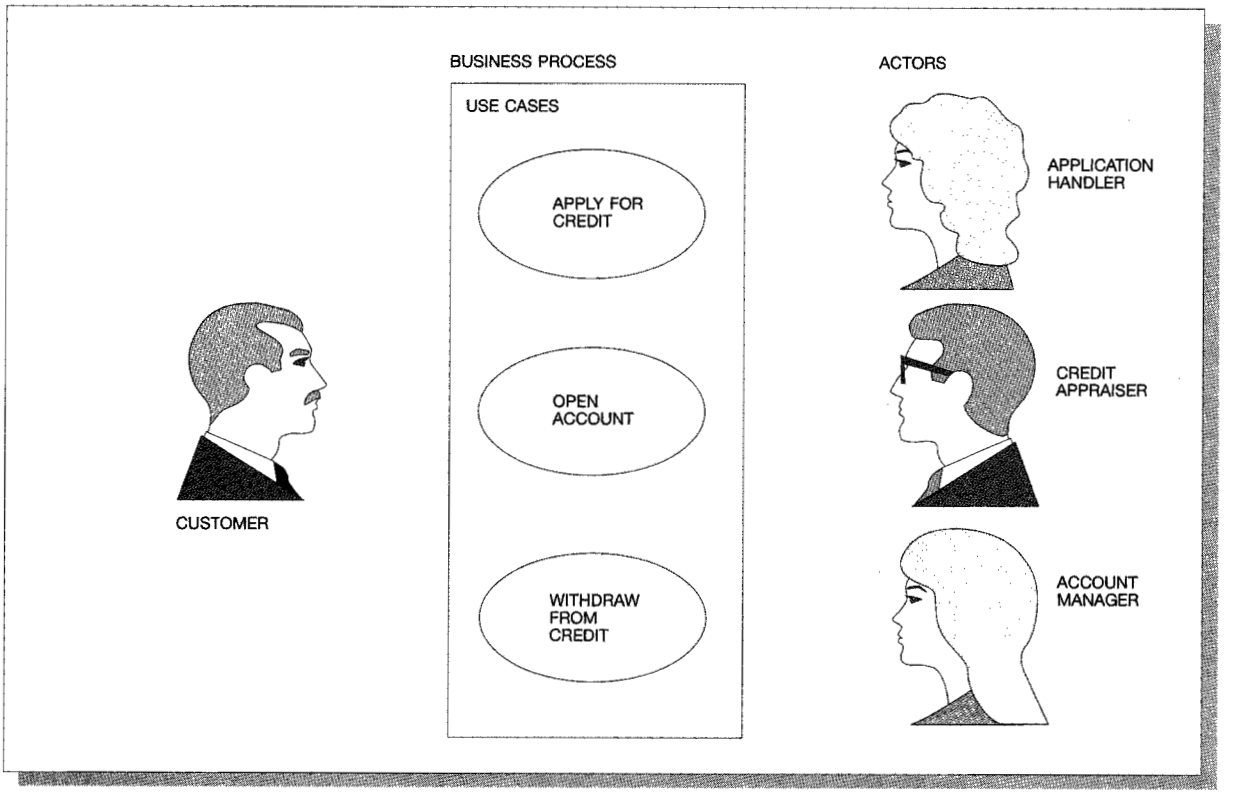
Figure 2 **Relating a business process to business objects. An external event arriving at a company triggers some part of a business process. This part of the business process is decomposed into a number of employee roles. Each employee role is decomposed into a sequence of business transactions to be performed in a workplace dedicated to that business process. Each business transaction must be processed by the information system and is decomposed into a series of messages. Each message invokes a specific service at one of the business objects in the system.**



tional requirements, into the specific business objects as needed. The specific business objects thus have a large set of properties in common. They display many common family traits, since they all belong to the same business information system. Exploiting these common traits, by reusing the generic object, has a very positive effect on both quality and productivity.

This development strategy is not limited to the nonfunctional requirements; it can be extended into the functional requirements. To support the flow of work in business processes, business objects are assembled into applications and these applications are in turn assembled into workflows. Just as for integrated circuits that are assembled into system boards and system boards that are assembled into computer systems, interfacing standards are needed. Finally, the method we used to model the functional requirements proved to be a source of genericity. Because we expressed the systems analysis results in a consistent way in our models, we found that the sub-

Figure 3 The business process is supported by use cases. For each external event that arrives, a separate use case describes the part of the business process that is triggered by the event.



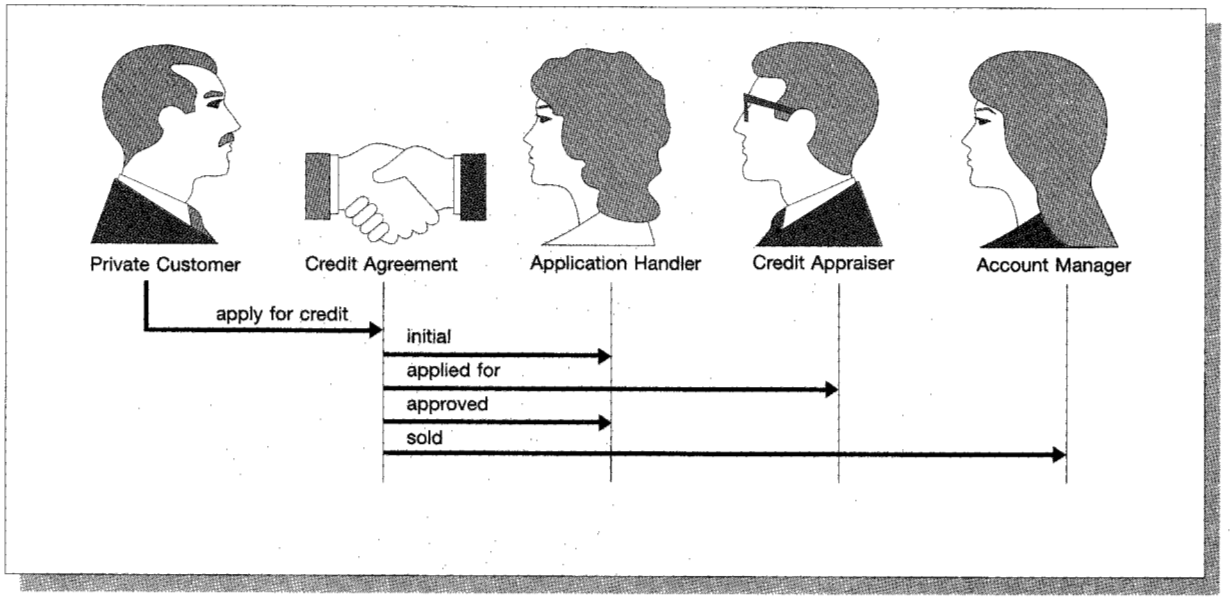
sequent design and implementation activities for different business objects had a common "meta" structure. We could thus make additional uniform design and implementation decisions early in the process and add these to our generic business object.

Business-object analysis

In our analysis we concentrate on what the information system must provide to assist the business processes; i.e., we concentrate on modeling the functional requirements. Our aim is to build an information system that mirrors the object-based, concurrent, and event-driven nature of a business organization exactly as it is in reality. We start by identifying the external events that trigger the business organization into action (Figure 2). The scope of our analysis is the boundary of the business process that needs the support of the information system. We work on one business process at a time. A business process is a set of related activities, related because they all work toward a common goal. Examples of business processes in Dutch banks are credit management, fund transfers, and securities trading. We start the analysis process with the use-case technique, as introduced by Jacobson.[1] For each external event that arrives at a business, a separate use case describes the part of the business process that is triggered by the event. A single business process may span multiple use cases (Figure 3).

Figure 2 shows the three different layers that we use to document the event sequences that occur. We use time-line diagrams for this purpose. In these diagrams the vertical lines represent the participating entities and the horizontal arrows represent the events that flow. Time flows from the top of the diagram to the bottom, so each event shown by a horizontal arrow occurs after the one above it. We found

Figure 4 The use case is decomposed into actor roles. The apply-for-credit use case spans several actors. The state of the credit-management business process is maintained by the credit-agreement object. Triggers for the various actor roles are defined in the form of events that are to be raised when particular states in the business process are entered.



these time-line diagrams valuable not only in the business object layer, but also in the application layer and in the workflow layer (Figure 4).

Use-case definition. A use case is a sequence of transactions that one or more actors perform in a dialog with the information system.[1] A use case is triggered by an external event and describes the entire scenario for dealing with that event. The business objects in the information system represent the real-world objects in the business organization. Changes in the real-world objects must be communicated to their counterparts in the information system. We call such a unit of change in the real world a business event and the corresponding impact on the business objects in the information system a transaction. (Various methods model a similar unit of change but call it by a different name: McMenamin and Palmer call it an event partition,[2] Rumbaugh calls it an event trace,[3] and Coleman calls it a system operation.[4])

We list the business events in the chronological order in which they typically occur in the business process. We identify the external triggers that start a sequence of business events. For one external trigger, e.g., a customer applies for a credit account, the complete sequence of transactions with the information system is represented as a single use case. If there are several triggers, e.g., the various customer actions over time on a credit account, then a separate use case is needed for each trigger. The business process example shown in Figure 3 consists of multiple use cases. Like a class, a business process has instances. In our example of the credit management business process, the business process instance is the complete collection of business events that are related to a single credit agreement.

Business-process life cycle. It often happens that for one instance of the business process two use cases must be executed in a specific sequence. In that case, a business-process state, which is set as a result of the first and tested as a precondition for the second, must be defined between the two use cases. If one use case needs the involvement of multiple actors and the actors must work in sequence, then business-process states must also be defined between the contributions of the individual actors. The use case is segmented into parts, each involving a single role. The external trigger for a multi-actor use case is decomposed into individual triggers for the actor roles. We call this sequence of triggers a workflow (see

Figure 5 The static object model. Here we show the business objects and the persistent associations between them.

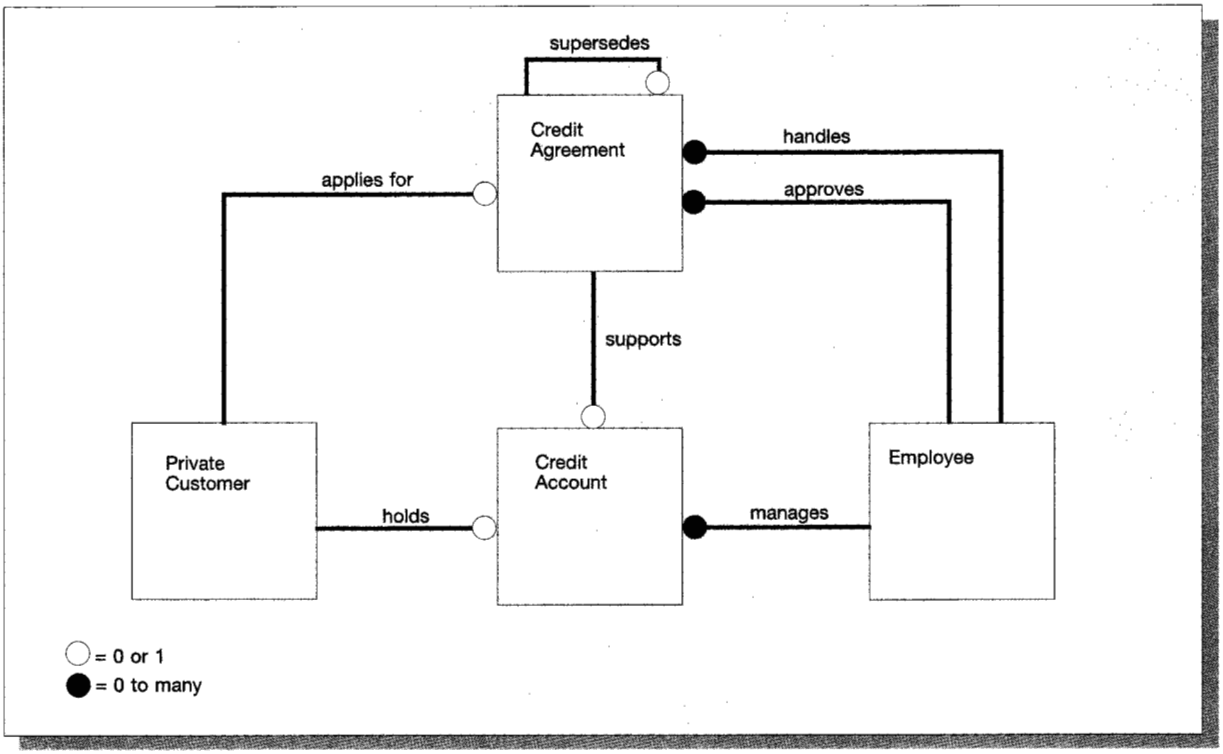


Figure 4). The trigger for each actor role is defined as the point in time that a business-process instance enters a particular business-process state.

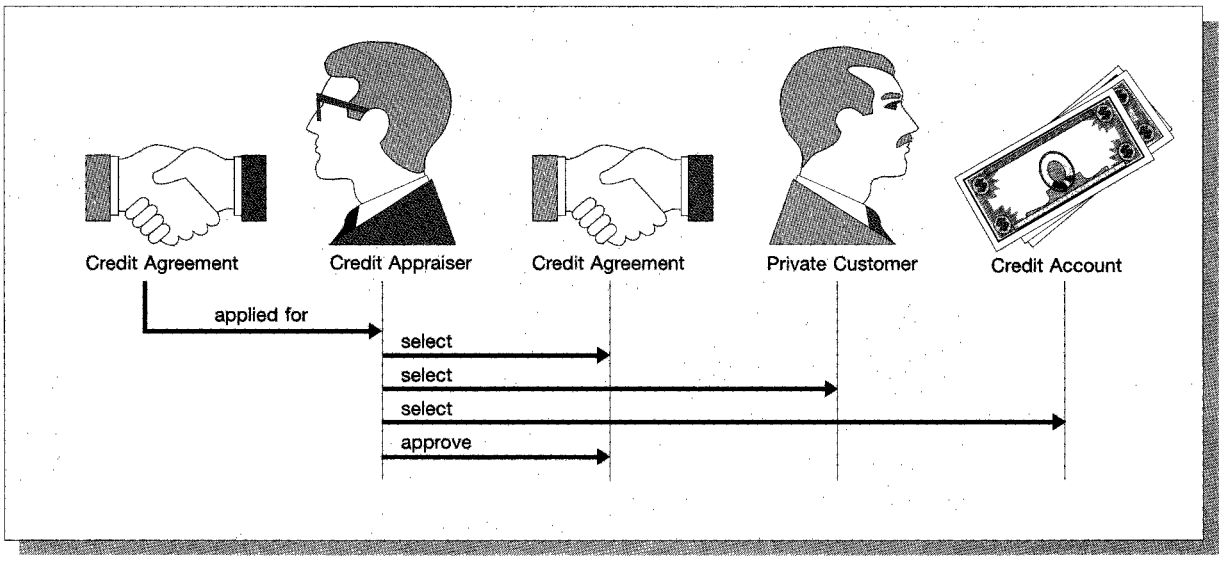
When we meet with the future users of the planned information system, we identify not only the use cases but also all business objects needed for these use cases. On a transaction-by-transaction basis we identify and list all the business objects involved. Usually one of the business objects we identify has a life cycle that corresponds with that of the business process instance itself. In our example it is the credit-agreement object. We assign to this object the responsibility to maintain the state of the business-process instance, in addition to its other responsibilities. It becomes the pivotal object of the planned information system for the business process and provides the continuous thread throughout that business process. It is a general requirement for any business to know what state any instance of a business process is in, so we never have to invent a special object; it always exists (e.g., a customer order, an insurance claim, a services contract).

Static object model. The business objects that were identified and the associations between them are drawn as a static object model,[3] as shown in Figure 5. The static object model describes the objects and the associations that persist over time. This static model diagram may also show inheritance relationships.

Dynamic associations, which exist only for the duration of a single transaction, are not drawn in the static object model. For example, an employee might execute transactions on business objects where there is no need to associate the employee permanently with the objects. Business objects already defined for the user interface, or auxiliary objects like printers and card readers, may be involved. For these cases a separate model is drawn to show the object population present on the workstation and from which dynamic associations may be obtained when executing transactions, as described later.

For each transaction in which a business object is involved, an action (implemented as a method) is

Figure 6 The role decomposed into transactions. The credit appraiser role is triggered at the moment that the credit agreement enters the "applied for" state. An actor playing that role will look at the credit agreement and the associated private-customer and credit-account data before approving the credit agreement.



defined that will create the local impact on that business object. For each object action the attributes are listed that are necessary to specify the result. The actions and the attributes are added to the static object model.

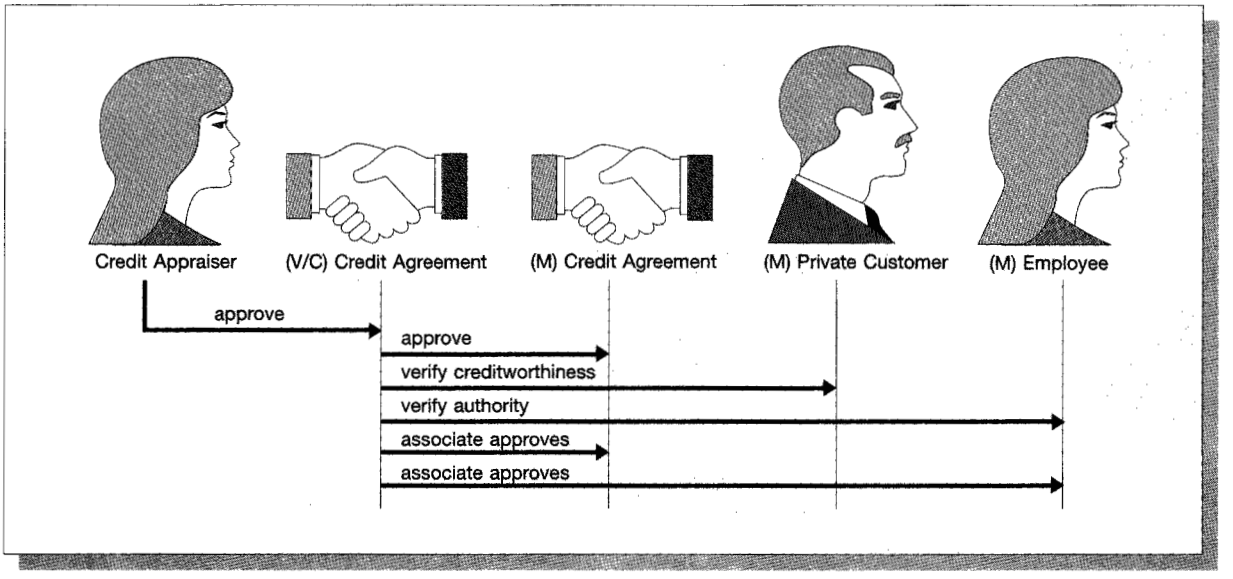
Our resulting system will consist only of objects, and all the operations on the system must be addressed to specific objects. So for each transaction that is listed in a use case, we decide the primary object, to which the transaction will be issued by the actor and through which the other objects in the information system will become involved. For example, a money withdrawal is issued to the credit-account object and not to the customer or till objects. We create a time-line diagram for each actor role in a use case by showing the role together with the business objects involved, drawing an arrow for each transaction starting from the role and leading to the object with the responsibility for the transaction. The trigger for the actor role is now decomposed as a series of transaction triggers to be sent to the business objects in the information system. This results in diagrams like the one shown in Figure 6.

Transactions. Just as we used a time-line diagram to show the decomposition of a use case into the actor roles, and later to show the decomposition of an actor role into the transactions, we use a time-line diagram here to show the decomposition of a transaction into the messages to be sent to the individual objects. This "object-interaction diagram" specifies what is required to process the entire transaction. It decomposes the transaction trigger into a set of separate messages that will in turn each trigger a discrete action on some business object.

Up to this point in our paper, we have treated a business object as a single entity. In our method a business object has several parts: a "model" part, representing its counterpart in the real world, a "view" part, providing the interface between the model object and the actor, and a "control" part. One business object will be held responsible for the complete transaction; it will accept the transaction from the actor through its view part and it will delegate the generation of the resultant message pattern to its control part. An extra vertical line is drawn for this primary business object in addition to the one that represents its model part; the second line is for its view/control combination. Additional vertical lines are drawn for the model parts of all the other involved business objects (Figure 7).

An arrow in the object-interaction diagram represents a message between two business objects. The

Figure 7 **The transaction is decomposed into messages. Each message will trigger one action against one business object. The primary object that receives the transaction from the actor has control responsibility over all business objects in the complete transaction, in addition to its responsibility for acting on its own data. The credit agreement is shown twice here, once for its transient view and control parts (V/C) on the actor's workstation, and once for its persistent model (M) part on the server.**



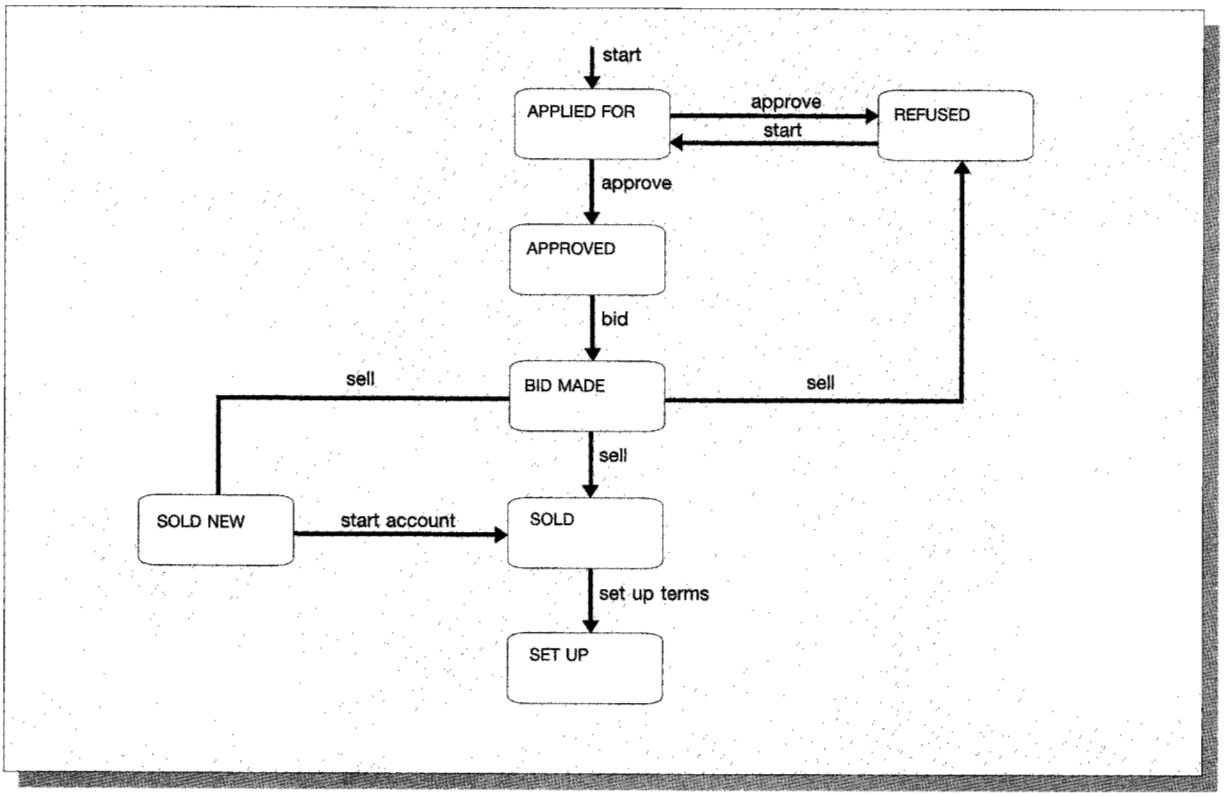
static associations specified in the static model and the dynamic associations specified in the workstation population are the available communication links between the business objects. With each horizontal arrow we list both the communication link (the association) and the message to be sent across it (the action to be invoked). During execution a transaction will be accepted by the control part of a business object only if all association instances that are mentioned in the object interaction diagram are available; otherwise a dynamic-binding exception will be reported to the actor.

The static model also specifies the multiplicity of each association and includes a minimum and a maximum number of instances for the association. As we specify transactions, we determine the number of times that they are executed during the life cycle of a single business-process instance. For each association, we look for corresponding multiplicities in transactions to determine which will create and which will delete an instance of the association. Actions to build and delete the association instances are then added to the transactions. In general we use bidirectional associations where one association always connects only two objects. Both halves of a bidirectional association are built during the same transaction, so both business objects must be visited.

Life-cycle model. After analyzing the transactions, we have a list of actions for each business object. We organize these actions in the sequence in which they typically occur during the life cycle of the business object. Usually we find that not all actions will be available at all times; the object changes its behavior over time. We assign a separate state name for each unique combination of actions supported by the business object during its life cycle. As mentioned earlier, we always assign the responsibility for maintaining the business-process state to one of the business objects: in our example, the credit agreement object. So for that object we have the states that are needed to sequence the use cases and the states that are needed to sequence the actor roles within the use cases.

For each behavior state, it must be decided which action, under what condition, causes the object to transfer to the state, and from what other state it comes. The life-cycle model of the business object is depicted in the form of a state transition diagram, as shown in Figure 8. (We also use the term "finite state machine model" to refer to this life-cycle model

Figure 8 The object actions result in behavior states. This example of a state transition diagram shows the initial states of a credit-agreement life cycle.



of the business object.) It represents each possible behavior state as a rounded box containing the state name. Each possible state transition is represented by an arrow that is accompanied by the name of the action that causes the transition.

Next we examine the list of actions supported by a business object and the list of behavior states that may occur during the object's life cycle. A matrix is created with the actions listed vertically and the behavior states listed horizontally. There is a row for each action, and all behavior states for which that action is supported are marked (Figure 9). It is often easier to determine the behavior states together with the user by creating this matrix first. In that case we create the state transition diagram later.

In our earlier description we left open the possibility for multiple use cases and multiple actors in the business process. If there are multiple actor roles and a separation of responsibilities between them is required, then the behavior states can be used to define that separation. For this we use the states of the business object responsible for maintaining the state of the business process—in our example, the credit agreement object. For this business object, the behavior states are extended with a states vs actors (roles) mapping in a matrix (Figure 10). Workflow logic can process the state transition events and, using this table, it can generate the triggers that activate the corresponding actor roles at the correct points in time.

Action modeling. Each action of a business object must be specified formally as a contract. Based on the transaction to which the object contributes the action, the precondition and the result of the transaction on the object is determined and the content of the action is modeled. (See Figure 11.) Five categories are explored:

1. The behavior-state precondition (finite-state

Figure 9 Mapping of business-process states to available transactions. The availability of the various actions depends on the behavior state of an object. The unavailability of an action implies that any transaction in which that action takes part will not execute. So a required sequence can be enforced by maintaining the business-process state under a well-chosen object and involving this object in all transactions of that business process.

ACTION \ STATE	INITIAL	APPLIED FOR	APPROVED	BID MADE	REFUSED	SOLD NEW	SOLD
start	X	–	–	–	X	–	–
approve	–	X	–	–	–	–	–
bid	–	–	X	–	–	–	–
sell	–	–	–	X	–	–	–
start account	–	–	–	–	–	X	–
set up terms	–	–	–	–	–	–	X

Figure 10 The roles responsible in the various states. Including business-process states enables us to enforce a separation of tasks and responsibilities within a business organization, and also to automate the propagation of a unit of work from one role to another.

STATE \ ROLE	APPLICATION HANDLER	CREDIT APPRAISER	ACCOUNT MANAGER
INITIAL	X	–	–
APPLIED FOR	–	X	–
APPROVED	X	–	–
BID MADE	X	–	–
REFUSED	X	–	–
SOLD NEW	–	–	X
SOLD	–	–	X
SET UP	–	–	X

Figure 11 **Each action needs a formal specification. In this simplified example there is one statement in each of the five categories. In general, each category may contain a single statement or multiple statements, or it may even be empty for some specific action.**

Object:	Credit Agreement
Action:	Bid
Finite-state predicate:	State equals "approved"
Business-data predicate:	BidExpiryDate < (ApprovalDate+90)
Data transformation:	BidMadeDate = TodaysDate
State transformation:	State ="bid made"
Event generation:	produce formal bid letter

predicate) is formulated with help of the matrix shown in Figure 9.

2. The data-state precondition (business-data predicate) is formulated (e.g., enforcing a withdrawal maximum or an expiration date).
3. The data-state transformation is formulated (e.g., updating a balance).
4. The behavior-state transformation is formulated and cross-checked with the state transition diagram.
5. A condition to be monitored may be formulated, in connection with an event to be raised if that condition occurs (e.g., a critical stock level is crossed when filling an order). Raising an event implies a new transaction, which must be modeled with an object interaction diagram of its own.

Each of these five categories may result in none, one, or many entries. Together they comprise the contractual specification of an action. If all the preconditions are met, then the object will be obliged to perform the action, otherwise it must remain unchanged.

As we model each action, we verify that all the attributes necessary to process the action are listed in the static object model. We decide which attributes are necessary for more than one action and so must persist between actions. We choose an initial value for each attribute. If an attribute value can be entered through the user interface, then we define a validation constraint to prevent contamination of the business data.

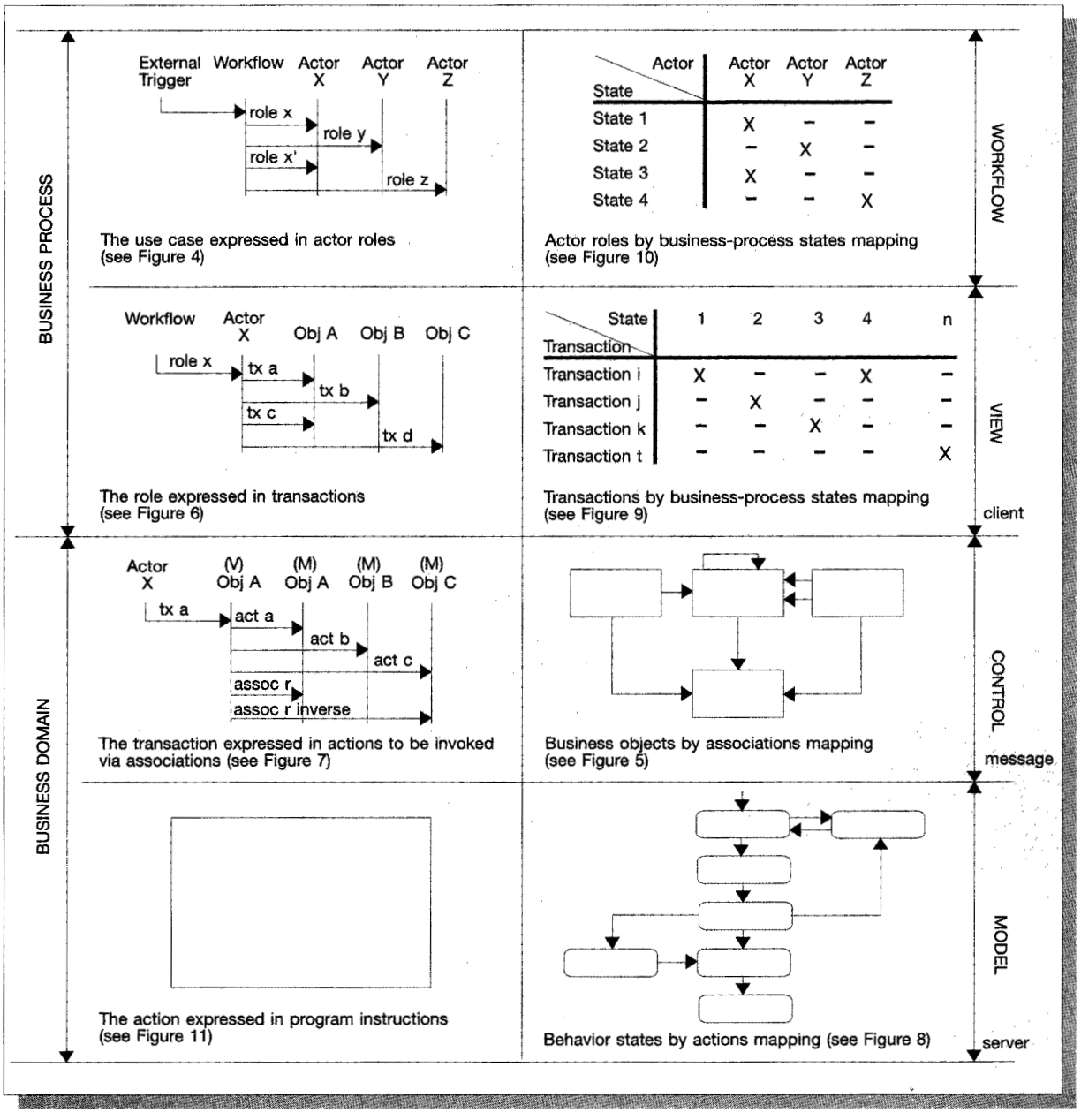
Iterate until stable. This straightforward, step-by-step description should not fool you. Reality is messy! We obtain the analysis products in raw form, with the future users, in a facilitated brainstorming session. We use Metaplan** for this front-office process (a good alternative would be joint application design [JAD] sessions). We complete this with a parallel back-office process in which we produce precise forms of these same analysis products, adding consistency and exactness to the requirements. Any omissions filled or assumptions made by the back-office process are fed back into the front-office process, iterating between the two until the analysis is complete. The result is a rapid application development process (RAD).[5] Each analysis product defines a relationship between just two dimensions of the target system. These products are simple enough to serve as the vehicle of communication between analysts and users, but at the same time they are exact enough to provide us with the information system specifications.

In the back office, when we draw a state transition diagram we may find an arrow (a transition) without an action. The action is added to the static model, the object interaction diagram of the transaction that contains the new action is updated, and the change to the business object resulting from the action is modeled. When modeling a transaction we may find an arrow (a message from one object to another) without an existing association. The association is added to the static model, as is the creation and deletion of the association through other transactions. When modeling an action we may find new attributes. This list of possible discoveries, followed by corrections, goes on and on. While applying our consistency rules, we are working on all of the analysis products concurrently. As a rule, the number of newly discovered inconsistencies converges rapidly to zero, and the analysis products are then stable enough to serve as the basis for the next phase: business-object synthesis.

Business-object synthesis

If the analysis is done properly, its products become specifications with enough detail to construct the business objects. None of the individual diagrams gives the total picture; they must be combined. Figure 12 provides an overview of the analysis prod-

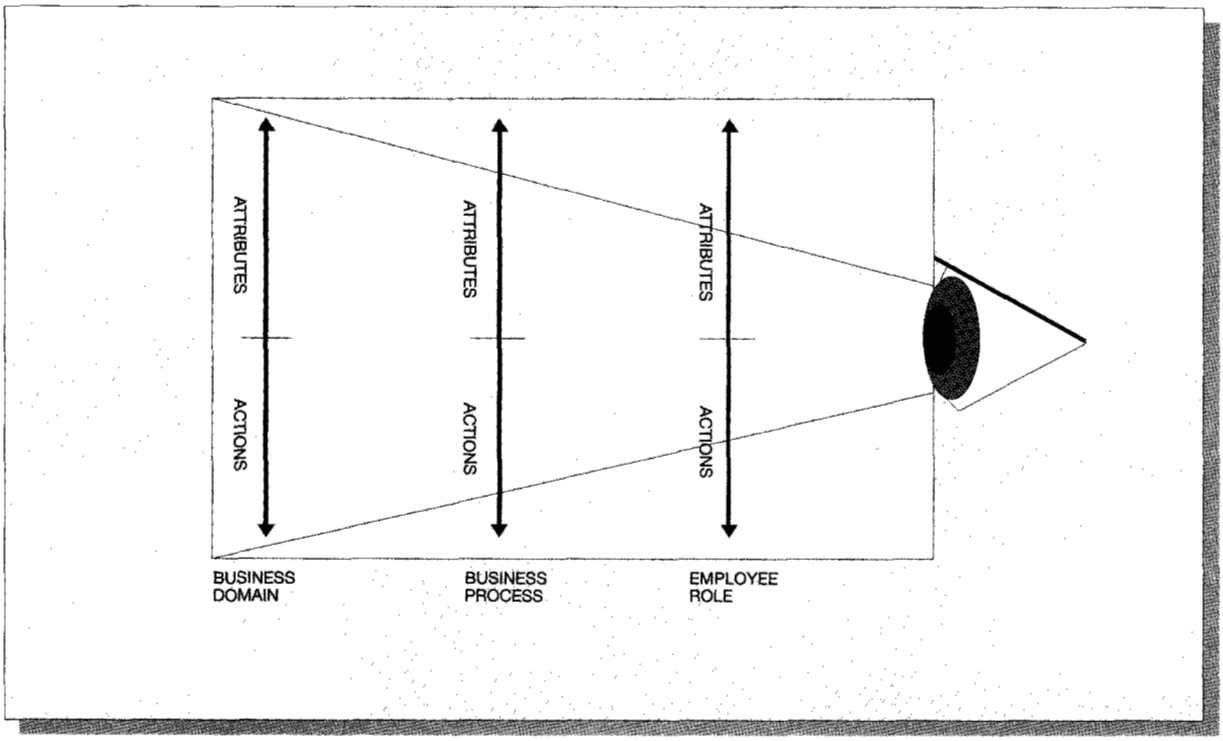
Figure 12 **The layered development model. The individual rows represent the development layers needed for business objects. The total schema illustrates the top-down analysis and the bottom-up synthesis process.**



ucts and maps them to a layered development process.[6] We now describe how they are linked together.

Our development strategy is based on business objects that are assemblies of four separate parts. Each of these parts manages a specific subset of the business object properties. The model part of a business object persists at a server. The view part, a proxy of the model, is visible and accessible to a user on a client workstation. Synchronization between a model part on a server and its view part on a client

Figure 13 Domain and business process coverage. An employee sees a subset of the attributes and the actions. The subset depends on the separation of tasks and responsibilities required by the business organization and the role assigned to the employee.



is maintained by a mediator between them, the control part. A business object that also maintains the business process state has an additional part that constantly monitors that state and actively coordinates the separation of actor roles and the necessary handoffs between them, the workflow part.

Model. We build our information system from the bottom up, starting at the server layer. This layer holds the model parts. The business-object actions are modeled as shown in Figure 11. For each business object, we collect all actions and order them in life-cycle sequence, as documented with a diagram as shown in Figure 8. These two specifications are sufficient to synthesize the model part of the business object. The model part interacts with services that provide persistence.

Control. We next build the transaction layer, which specifies all collaboration patterns between business objects and holds their control parts. It accepts transaction triggers from the client layer above and decomposes them into message triggers for the server layer below, as specified by the object interaction diagram as shown in Figure 7. The control part embodies the transaction services.

In each object interaction diagram are a number of model objects, each represented by its own vertical line. The control part of the business object needs the address of each involved model part in order to perform the required dynamic binding. These addresses are maintained in associations. We find associations in the static object model, as shown in Figure 5. From it we obtain all the associations that this business object maintains. We collect all the transactions that are triggered from this business object and combine them with its associations. From these two specifications we synthesize the control part of the business object.

View. We next build the client layer, where applications run that exploit the services of business objects. This layer holds the view parts. A user may directly manipulate business objects through their view parts, as displayed on a workstation. Countless ap-

plications are enabled by a display filled with business object views. Each different navigation pattern is equivalent to a separate dynamic application available to the user.

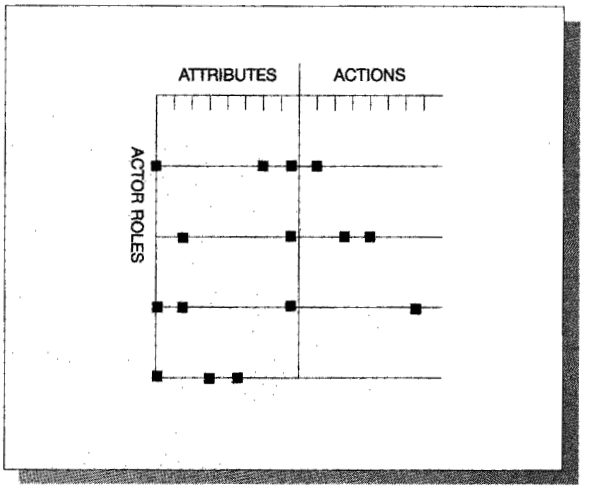
The view part contains the user-interface specification of the business object. To synthesize the view part, we reuse the actions and attributes specified for the model part. The view part restricts its behavior toward an individual actor by presenting only the subset of attributes and actions for which that actor is authorized (Figures 13 and 14) in cooperation with security services. All attributes are presented in a read-only window. For each action type, the view part provides a "dialog box" where attribute values may be entered. Even actions for which the actor is authorized may be rendered unavailable because of the current behavior state of the displayed object instance (Figure 9). The view part interacts with the presentation services of the underlying operating system.

An alternative form of the view part is available that uses SOMobjects* (SOM). It provides the same attribute and action interfaces through the Common Object Request Broker Architecture** (CORBA**) -compliant Interface Definition Language (IDL) interface to a client application. This form bypasses the presentation services, but uses all other framework facilities. So the same security services, transaction services, persistence services, workflow, etc., will be mobilized by the framework on the inside, but remain transparent in the SOM interface. In this alternative form we have successfully imported our business objects into a VisualAge Smalltalk application. This approach is especially promising in places where safeguards for business data integrity are managed centrally while applications can be constructed as close as possible to the user department.

The procedure for developing the view part for one business object has just been described. In the role description shown in Figure 6 there were several business objects, each represented by its own view part. The procedure is followed for each business object. If an application must be built on top of the business objects, the role description becomes the application specification.

The business object's support for a particular business event can now be defined as the summation of the corresponding dialog of its view part, the corresponding transaction of its control part, and the

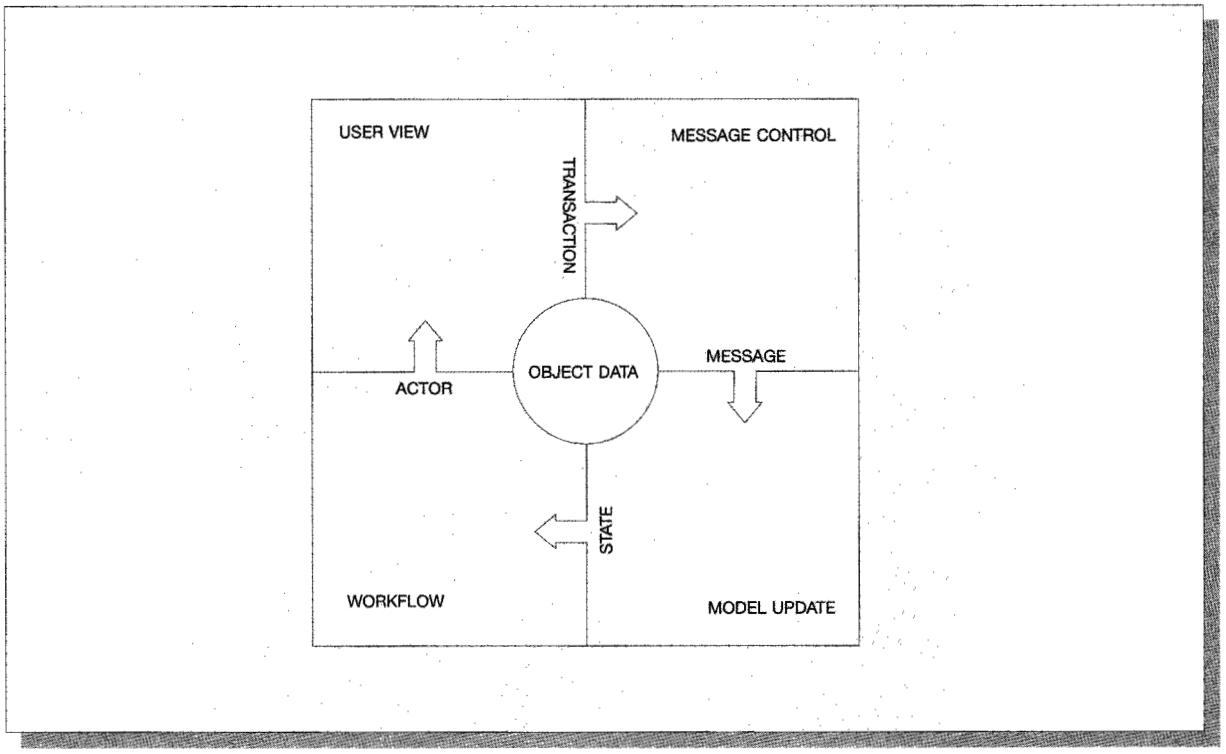
Figure 14 **The business object enforces security. It knows all actor roles and the actions and attributes that are available to those roles. The view part inspects the role of the workstation user and dynamically modifies its behavior accordingly.**



corresponding action of its model part. The business object must support all business processes in which it is involved; thus it may contain more actions and attributes than are required for a single business process. The business object in its broadest form we call a "business domain object." The view part of a business object has complete business domain coverage; since it runs on the client workstation it also knows the employee role that it supports. A "filter definition" dynamically specializes a business object for each specific actor role. Its view part uses the filter to adapt the behavior of the business object to the business-process context in which its services are used. In this way, one business-object type emulates the whole family of closely related object types that would otherwise be necessary.

Workflow. The highest layer of our development process is the workflow layer. If there is a separation of tasks and responsibilities within a single business process, then a workflow construct becomes necessary to trigger the various actors at the right moments in the business-process life cycle. In a business process, we always choose one business object that serves as the continuous thread. For a business process selling products, it could be the customer order. For a business process providing services, it could be a contract. For any business process, we choose the business object that embodies the work-

Figure 15 Final assembly of the business object—the CYCLADE building block

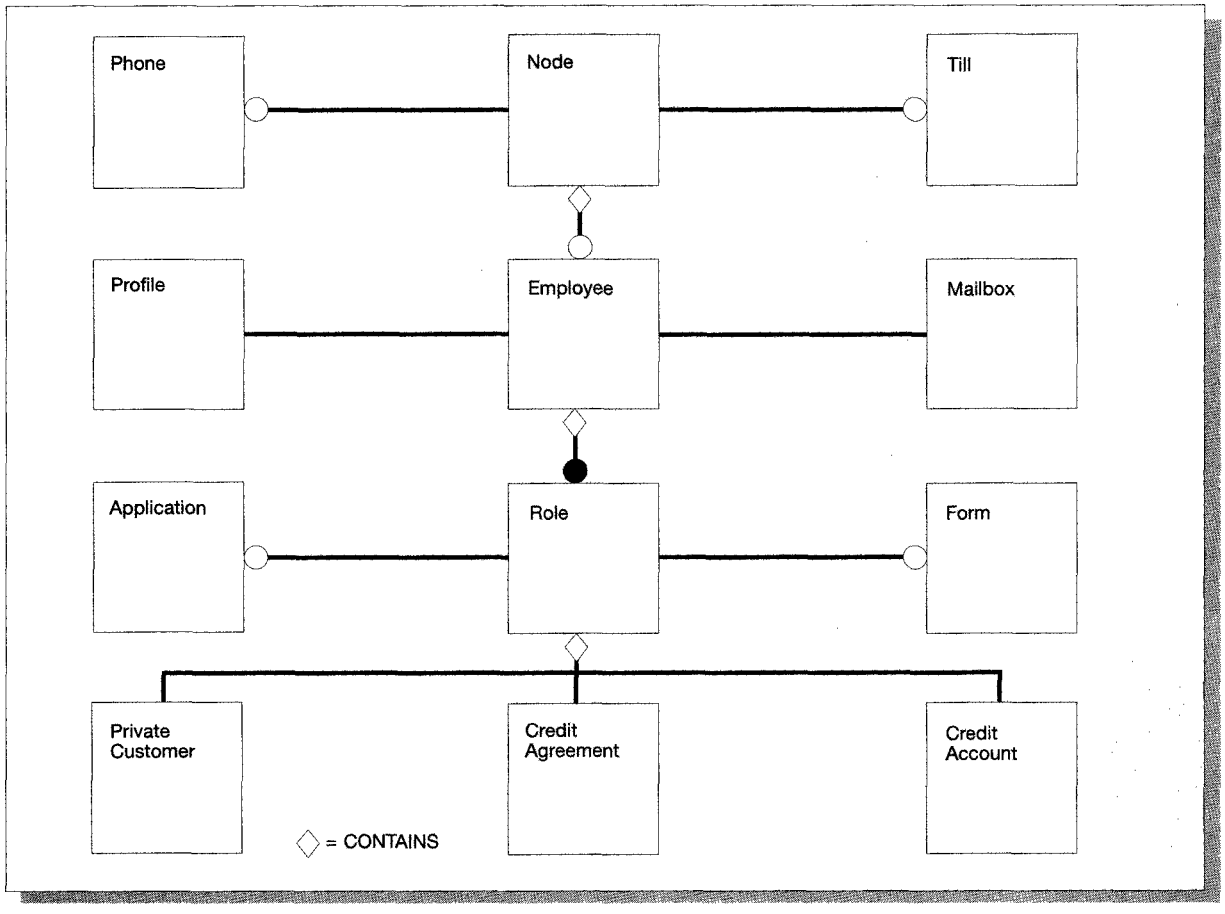


flow. In our example, the workflow is attributed to the credit-agreement object. The workflow part of this business object is notified whenever the behavior state of the credit agreement changes, and it distributes work over the various business-process roles according to the states-vs-actor-roles matrix (Figure 10).

As already mentioned, the view part adapts itself to the actor role that it supports. We found a similar dynamic to be necessary for the workflow part. Companies with multiple sites may have a large population with a deep specialization of actors at one location and a small population of generalists at another location. The same credit-management business process may, for example, demand a different workflow at a small-town branch than at a big-city branch of the same bank. So we have a workflow definition that can differ for each company location and still be part of the same business object. A huge number of variations is needed within the same business-object type to accommodate its presence in different business processes, in different workflows, and in different actor roles. This variety results in a large family of closely resembling, but slightly different members. The desired variety is created by dynamic interpretation of customization parameters. The specification of these parameters does not require development expertise and, given a proper tool, may be performed by administrators.

After we synthesize the four specific parts of the business object—model, view, control, and workflow—they are assembled into the complete business object as shown in Figure 15. The picture is deliberately drawn as a framework. It is also drawn as a jigsaw puzzle. We create the business object in various pieces and then assemble it together. The view part maps the actor to the available transactions, the control part accepts a transaction and maps it to the messages for the involved model parts, the model part maps the incoming message to the resulting state, and the workflow part maps the resulting state to the effecting view. This uniform pattern is incorporated into the generic business-object form that we constantly reuse. So an employee may feed a bus-

Figure 16 **The workstation configuration. This is a simplified example of a teller workstation. The workstation is connected to a telephone and is associated with a till. When an employee logs in, the workstation becomes associated with that employee, and a population of additional business objects is rolled in. For each particular role of the employee, a specific assembly of business-object proxies is started. The resulting composition of business objects will also be the source of dynamic associations when transactions are started by the workstation user.**



iness event to the information system by invoking a transaction from a business-object view. As soon as the employee releases the transaction, the business object will take a spin along its various components and it will come back to the actor with a fresh view representing the updated state of the object. This uniform pattern turns each of our business objects into a CYCLical Application Development Entity, so we labeled it the CYCLADE building block.

The business object has become an independent development entity through addition of a unit-test mode. If it is triggered by its view part while running in unit-test mode, then the control part will not act as a mediator for the other involved business objects; instead it will activate only its own model component counterpart. We can thus build and test the entire business object as a stand-alone entity and integrate it later with the other business objects. This is the key to component-based development from functional requirements. We could not have done this without our particular choice of responsibilities for the control part of the business object. It sets us somewhat apart from other approaches. Keeping the attributes of a business object and all the operations on them in one model part is universally accepted as an object-oriented principle. Keeping the presentation of a business object's attributes and all the di-

alogs for accepting operations on them in one view part is equally accepted as an object-oriented principle. Keeping the associations of a business object, and all the messages across them, in one place is comparable in reasoning. We think this is also an object-oriented principle, and we introduced it as such.[7]

After the individual business objects are assembled from their component parts, there is one remaining step. The business objects now must be assembled together in the combinations needed to support the various business-process roles. We prefer to make a separate assembly of business objects for each role. Sometimes we hide the loosely presented business objects behind an integrated application that interfaces to the same underlying transactions but exercises them in fixed patterns. A compound-document interface like OpenDoc* may be used, in a similar way, to hide the individual business objects behind a single form and to overlay them with a fixed application script. An employee may have multiple roles and each of them must become enabled as soon as the employee has logged on to the information system (Figure 16).

Project experiences

Earlier object-oriented-development projects in manufacturing and in banking environments led us to the insight that a framework approach to business objects would be a powerful way to develop information systems. During our research period we gathered additional insights from projects that our customers were developing.

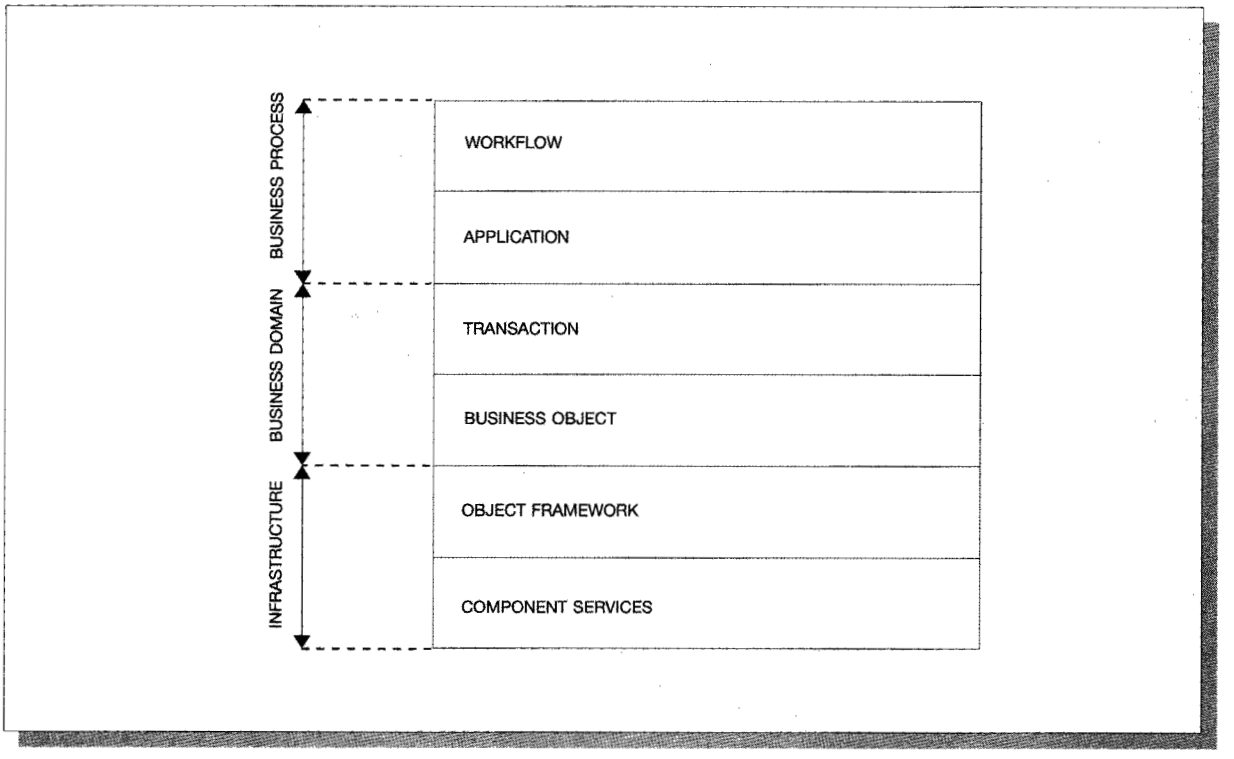
The first lesson we learned was to abstract the communication responsibility away from the other responsibilities of the business object. Several of our Dutch banks have proprietary communication protocols that they use across nonstandard network infrastructures. From an external supplier, they expect solutions that will not only run in their homegrown environment but in addition will avoid obsolescence by supporting market standards like CORBA. So in a future system, a business object may need to collaborate with other business objects inside the same local domain, through a Smalltalk reference or a C++ pointer, or inside a remote domain, through a CORBA, an OSF/DCE** (Open Software Foundation's Distributed Computing Environment), or even a proprietary reference. We decided to let a business object work with an abstract reference to other business objects and embodied that reference in the association. A message layer was now sitting on top of the business objects that decoupled them from each other and was responsible for organizing the collaboration of the right combinations of objects in specific domains.

The second lesson we learned was to abstract the application context away from the other responsibilities of the business object. The Dutch insurance company considered to be the leading-edge applier of object technology in the Netherlands was having problems extending the use of business objects beyond their first application contexts. If nothing is planned above the domain level of the business objects, then every business rule has to be allocated to that single business-object model. Business rules representing business-process properties then creep into the business-object domain representations. Reuse of these business domain objects in other business processes then necessitates changes, which must be analyzed with respect to their impact on the earlier applications. More often than not, such domain objects cannot be reused, because they are too dependent on their original context! So we added a third layer to hold the application-specific rules.

The third lesson we learned was to abstract the different user roles away from the domain responsibilities of the business object. Businesses generally organize a separation of tasks and responsibilities into their business processes. So there are also security and workflow considerations to be taken care of. With the present focus on business-process re-engineering, the empowerment of employees changes over time. But even at a single moment in time there is considerable variation. One of the banks we worked with was organized as a federation. Each member bank shared the same products but had its own business-process responsibility. So we added a fourth layer to hold the rules for the specific work distribution.

At first sight, layers look complicated. On closer inspection they are often justified as powerful complexity-reducing mechanisms. Complex products like computers, cars, homes, etc., are produced in layers. Consider computers: producing integrated circuits comes first, assembling these onto printed circuit boards is next, and finally the printed circuit boards are assembled into computers. Identifying several successive layers of assembly enables the production process to be organized as a chain, with value added with each link. In this chain the products flow in one direction and the money that symbolizes their

Figure 17 The layered development approach



demand flows in the opposite direction. Components produced in earlier stages are often reused later for new products not even anticipated when the component was first designed. When the demand stops, at any point in the process, the money stream runs dry and all downstream production processes go out of business.

Application development is a complex production process, and we found layering to be equally beneficial here. Once we visualized the development layers as shown in Figure 17, we realized that we had defined a component-based development strategy. In other disciplines, component-based production processes converge in ever-bigger assemblies until some external demand is met. So it was important for us to ensure that there was a top-down stream of derived demand for the products of the bottom-up assembly process. This was achieved via the stepwise decomposition of each external demand until the level of computer-program instructions was reached. As described in the analysis section, we decomposed each external demand into an assembly of actor roles, each actor role into an assembly of transactions, each transaction into an assembly of messages invoking business-object actions, and each business-object action into an assembly of individual program instructions. These decompositions specify the dynamic properties for the various layers. The other analysis products model the static properties of the various layers.

End users without any prior exposure to application development were quite able to talk to us in the terms of the analysis model. It seems to be very close to the way they themselves mentally model the business process. The analysis model also proved to be an excellent vehicle for communicating with the developers. We had already developed an object framework to accompany our analysis framework. Development of each business object, using the object framework, proved to be atomic; each object was developed and tested independently. The object framework successfully integrated the separate objects by interpreting the static object model, the object-interaction diagrams, and the various state diagrams. Only the model, or server layer, had to be programmed; all higher layers were directly derived from

the standard analysis diagrams. With the analysis and synthesis steps we have not only covered all business requirements, we have also achieved an implementation form that can be easily verified against the requirements.

The framework approach was chosen to help implement the business requirements rapidly by executing the majority of the analysis models directly. The techniques as described in the object analysis and synthesis sections were selected first, and the object framework was developed next to exploit the outcome of these analysis techniques. So the techniques are valid in their own right and can be applied without having access to an object framework. In our object framework the view part, the control part, and the workflow part of the business object are fully generic. They interpret the analysis products as given in the upper three layers of Figure 12. The model part is abstract and must be subclassed for each business-object type. With our approach, even very different business objects have more than 90 percent of their code in common. They are closely related and share many family traits.

This object framework adds another layer of its own. In fact, the full object infrastructure adds two layers, separating the implementation of the nonfunctional requirements from the business requirements. Implementing the platform-specific properties in a separate layer of component services not only enhances productivity, it also provides heterogeneity and portability. Rewriting the component services layer for another platform allows everything above it to be migrated without change. We have found that layering provides us with predesigned reuse. We started with an OS/2* (Operating System/2*) implementation, and with little effort have added Microsoft Windows** to the client environments and AIX* (Advanced Interactive Executive*) to the server environments and can run our business objects and applications in any combination of these environments (Figure 17).

A frustrating property of software development is that when you have learned to produce a solution in a given environment, the environment changes and your solution becomes outdated. Now that we have an object framework for distributed business objects written in C++, the emphasis for distributed objects systems seems to have suddenly shifted to Java**. Recently we performed an assessment of the effort needed to migrate the object framework to a Java platform. We concluded that our development process is needed just as much in that environment, and that migration of the software is feasible. We can easily see the benefit, a single version of the object framework would then suffice for many hardware platforms. The conceptual integrity of the combined process and framework becomes even more important in a Java environment. Ensuring the secure access and transaction integrity, combined with a flexible user interface and an integrated workflow, striking the right balance between centralized security and decentralized development will become a matter of survival in that powerful new environment.

We have only recently concluded our research. To obtain a "proof of concept" we have developed two pilot projects with customers in the banking field (the approach is valid for businesses in general and is not limited to banks, but the first volunteers happened to be banks). Our conclusions are quite positive. A systematic approach accelerates reuse. Complexity is reduced: the consistency of the combined set of analysis models and the many design decisions encompassed by the software framework make it much easier to focus on the business requirements. Modeling becomes far more important than coding; it is both a better communications vehicle and a more productive approach to development.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Metaplan GmbH, the Object Management Group, the Open Software Foundation, Microsoft Corporation, or Sun Microsystems, Inc.

Cited references

1. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley Publishing Co., New York (1992).
2. S. M. McMenamin and J. F. Palmer, *Essential Systems Analysis*, Prentice Hall, Englewood Cliffs, NJ (1984).
3. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ (1991).
4. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, NJ (1994).
5. A. Blokdijk and P. Blokdijk, *Planning and Design of Information Systems*, Academic Press, London (1987).
6. R. Prins, *Developing Business Objects: A Framework Driven Approach*, McGraw-Hill International Ltd., Maidenhead, Berkshire, UK (1996).
7. R. Prins, "Beyond Transactions: A World of Interworking Business Objects," *First Class* **5**, No 2, 16–20, special issue on Business Object Management (1995).

Accepted for publication August 15, 1996.

Robert Prins *IBM Consulting Group, Europalaan 440, 3526 KS Utrecht, The Netherlands (electronic mail: rob_prins@nl.ibm.com).* Mr. Prins is a consultant with the Application Development Effectiveness Consulting Practice in Utrecht. He joined IBM in Amsterdam in 1969 as an application programmer. In 1974 he became an instructor, teaching structured programming and design. In 1978 he moved to industrial automation, and in 1984 he became the architect of a new plant floor concept that was one of the earliest large-scale applications of object-oriented techniques. When the IBM facility in Amsterdam was closed in 1987, he became an industrial consultant. He has worked for the Application Development Effectiveness Practice since 1989 as a specialist in object technology. In addition to consulting with Dutch and Belgian companies, he teaches at IBM's Object Technology University in La Hulpe, Belgium.

André Blokdijk *IBM Consulting Group, Europalaan 440, 3526 KS Utrecht, The Netherlands (electronic mail: andre_blokdijk@nl.ibm.com).* Mr. Blokdijk is the practice leader of the Application Development Effectiveness Consulting Practice in Utrecht. He joined IBM in Amsterdam in 1967 as an analyst/programmer. In 1971 he moved to IBM Education, where he developed and taught courses on structured development techniques and project management in conjunction with consulting engagements for large customers. In 1983 he became a faculty member of the IBM European Systems Research Institute (ESRI) in La Hulpe, Belgium. His research was on creating a framework for comparing development methods and on rapid application development techniques. He developed ESRI courses for consulting on information strategy planning and on application and data planning. This resulted in many lecture and consulting engagements in Europe, the Middle East, and Latin America. In 1989 he became a director of Cyclade Consultants, then a joint venture with CAP Volmac, now fully owned by IBM and part of the IBM Consulting Group. Since 1988 he has been chairman of the examination council on methods and tools for information system development, EXIN, which is the Dutch national examination institute for information sciences. He is a member of the NGI (Nederlands Genootschap voor Informatica), the ACM (Association for Computing Machinery), and the Computer Society of the IEEE (Institute of Electronics and Electrical Engineers).

Norbert E. van Oosterom *IBM Consulting Group, Europalaan 440, 3526 KS Utrecht, The Netherlands (electronic mail: oosterom@nl.ibm.com).* Mr. Van Oosterom is an associate consultant with the Application Development Effectiveness Consulting Practice in Utrecht. He received his M.S. degree in computer science from the University of Nijmegen in 1990. Before joining the IBM Consulting Group in 1995 he worked on new technology transfer, including object technology, between universities and businesses in the Netherlands. He divides his time between teaching at IBM's Object Technology University and consulting for large customers.

Reprint Order No. G321-5632.