Technical note

Business function specification of commercial applications

by D. Bevington

Commercial application development projects frequently take longer and cost more than their sponsors would wish. One problem area is an uncomfortable "join" between the work of business analysts, responsible for understanding and specifying the business function required, and programmers, responsible for designing and building the implementation on a particular hardware or software platform. Proposed here is a small set of analysis constructs that can be used to specify precisely the business function that a commercial application must implement. The constructs cover data, processes, and user interfaces, but are free of platform-dependent implementation detail. The set of constructs is based on a common structure shared by commercial applications regardless of the business function being implemented. Instances of the constructs can be created to completely specify the business function of a particular commercial application. A separate transformation process can then take place to implement the business function specified on a target platform. The transformation required does not depend on the business function specified, but on which construct is being transformed, on nonfunctional requirements such as performance and security, and on the architecture of the target platform. The constructs allow a better separation of design concerns and provide more precise and complete communication between business analysts and programmers.

During the 1980s and 1990s, most large- and medium-sized enterprises built a set of commercial applications that are critical to the day-to-day running of their businesses. These applications cover fundamental business functions, such as customer records, order entry, billing, inventory control, and financial records. They started as simple productivity aids but have steadily expanded in scope and function. Today these systems are frequently in urgent need of replacement for the following reasons:

- They are inflexible and severely limit the ability of the business to enter new markets, develop new products, and transform themselves in various ways that are appropriate to the environment in which they now find themselves.
- They are very expensive and difficult to maintain.
- They are based on obsolete technology and cannot easily exploit new technology, such as graphical user interfaces and the Internet.

Despite the promise of new approaches such as object technology, many of the attempts by enterprises to replace these critical legacy systems have not been successful. This has led to an increasing use of packaged solutions and to reusable frameworks such as SanFrancisco, IBM's set of commercial frameworks. While this is an appropriate direction in many cases for many enterprises, most enterprises will still require the ability to develop their own customized applications for some areas. They must therefore address the problems of commercial application development.

Although there are a number of reasons for application development projects being less successful

than developers would wish, just one particular problem is addressed here. This is a significant problem and successful resolution of it will have benefits across the application development life cycle. However, it should be emphasized that there are many other aspects of commercial application development that must be managed correctly to ensure success.

Commercial applications

Commercial applications follow a very specific pattern, whatever their particular business function. They are based around one or more databases, which are shared by many users. Each database reflects the current state of some business domain, and serves as the primary set of business records for the domain. Function is provided to users to support business operations, and this function is dependent upon the state of the domain databases. As business operations progress, the databases are in turn updated to reflect the old state that is now history, the current state of the business domain, and planned future states.

The databases involved are the fundamental records of the business, and their integrity and correctness are vital. Concurrent update by multiple users has the potential to destroy this integrity, so controls to guarantee integrity are a fundamental part of such applications. Note that this requirement has nothing to do with computers. It is a fundamental logical requirement, caused by allowing multiple users to update a single shared master record. The same issues arise with systems based on shared folders or ledger cards. Fundamentally the business has to make a decision as to when some updates to a record by one user cease to be restricted to a private copy that can be discarded, and instead are reflected in the shared master record. The updates then become visible to all users and can be the basis for further actions by them. In other words, the business must define logical units of work (LUW), and this definition is a key part of the definition of the business function of a commercial application.

The analysis problem

Typically there are two main groups of people involved in the building of a commercial application: the business analysts whose skills are in understanding the business requirements and designing the business function to meet them, and the technical experts whose skills lie primarily in system architecture and programming. In an ideal world everyone involved in building an application would have both skill sets, but in practice this is seldom possible.

Problems frequently arise because business analysts make two critical mistakes: (1) they fail to fully define the business function required, and (2) they express some business function design in terms of a particular implementation design. As a result, the technical experts attempt to resolve these problems by: (1) completing and reworking the business function design, without the necessary wider business awareness to fully understand the implications of their decisions, and (2) reworking much of the implementation design done by business analysts. The result is systems that are not as flexible as the business requires, and that take much longer to build than necessary.

Surveys done by IBM as part of the Enterprise Solutions Structure (ESS) project[1] indicate that this problem is both common and a major cause of application development projects not delivering systems that meet the expectations of their sponsors. The problem is illustrated graphically in Figure 1.

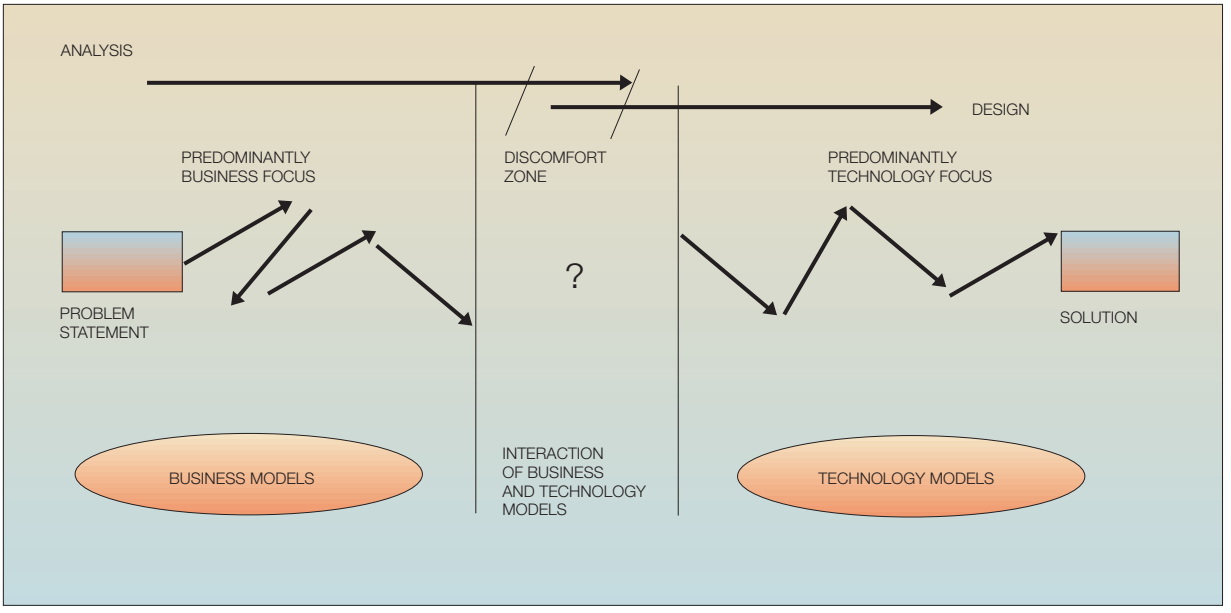
The business function specification requirement

The characteristics of a commercial application mean that an analysis model comprising an undifferentiated collection of classes and their methods is not sufficient to describe the business function required. The implemented systems comprise a diverse set of objects—user interface objects, persistent database objects, workflow procedures, database managers, transaction managers, etc. Accordingly the definition of the business function of an application needs a more extensive set of constructs that are platform-independent abstractions of the very different components found in the implementation.

The constructs proposed here provide the necessary abstractions. They could be considered metaclasses,[2] classes whose instances are themselves classes. In the following sections they are defined informally and illustrated with informal diagrams.

The constructs incorporate the notion that it is possible to abstract the business function of an application to a level where data, processing, and user interaction are specified in a way that is independent of technology. Business function can be implemented using pencil and paper, ledger cards, and the like,

Figure 1 Breakage in the application development process



as well as with computer systems. The constructs capture the business function specification, and presume a separate activity that is the mapping of the business function onto a specific hardware or software platform. This is illustrated in Figure 2, which also highlights that they are two further major inputs to the final platform implementation. The first set consists of nonfunctional requirements (performance, availability, security, etc.), and the second of technical considerations that arise from the particular characteristics of the implementation platform chosen.

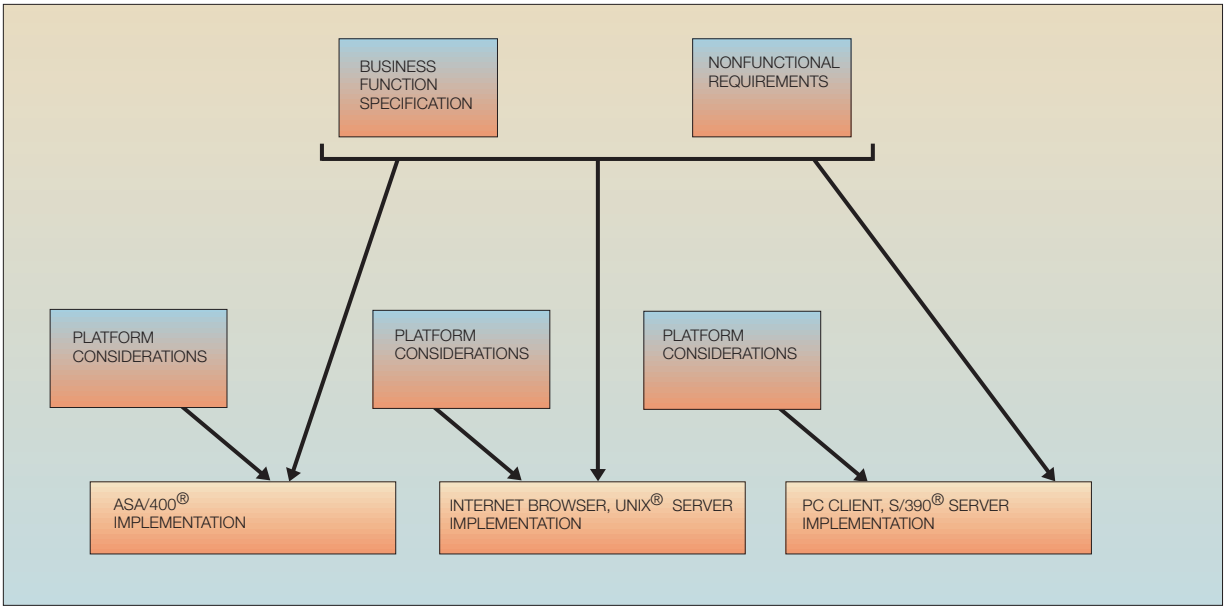
The proposed constructs help to resolve the problems covered earlier. They can provide much better communication between business analysts and platform implementers. They also provide a template that encourages completeness of the analysis. While the template does not force full definition of the very fine detail of some behavior, it does force the business analysts to properly consider many areas that are typically neglected. Most critically, it forces the business analysts to define the LUWs, which are fundamental to the design of this class of applications. It also prevents analysts from getting involved in platform design that may be inappropriate. Analysts do so because they have no other way of expressing some of the function required. With the abstractions provided by the proposed constructs they can express this function in a platform-independent form. This provides a much better separation of design concerns, leaving the platform design to be done by the appropriate specialists, and easing implementation of the same function on multiple platforms.

The proposed constructs

The constructs are defined here using a simple natural language definition. Note that there are alternative sets of constructs that could have been proposed. The set proposed here is designed to simplify the transformation from business function specification to platform implementation without unduly compromising the degree of platform independence of the business function specification.

Accompanying the definitions are some informal diagrams designed to illustrate the relationships between the constructs. A simple application is also used to provide particular examples of an application specification using the constructs. The example specifications use natural language but with some consistent syntax and layout to make the specification more precise. A more complete specification of this example application is available.[3]

Figure 2 Mapping of requirements to platform as a separate activity



The example used is a simple education center application. The education center teaches a range of courses, and particular offerings of these courses are run on particular dates. Students enroll in offerings of the courses, and then attend them. Each course offering has an instructor who is responsible for teaching the course. The application provides for the recording of courses and course offerings. It allows students to be enrolled in an offering of a course. When the course offering has been run, completion can be recorded along with the grades achieved by the students. A Unified Modeling Language (UML)[4] static object model of the persistent data of the application is shown in Figure 3.

Shared persistent business objects and related constructs. The constructs include the common notion of business objects, which are objects representing the data and behavior of the business entities that the application is intended to manage. More precision is added by making it explicit that these objects are persistent and shared by many users. They are differentiated from other objects that are local to a single user or are not persistent. The constructs are defined as follows.

Database. A database is a collection of shared persistent business objects, events, and transactions, where the transactions only access shared persistent business objects that are part of the database.

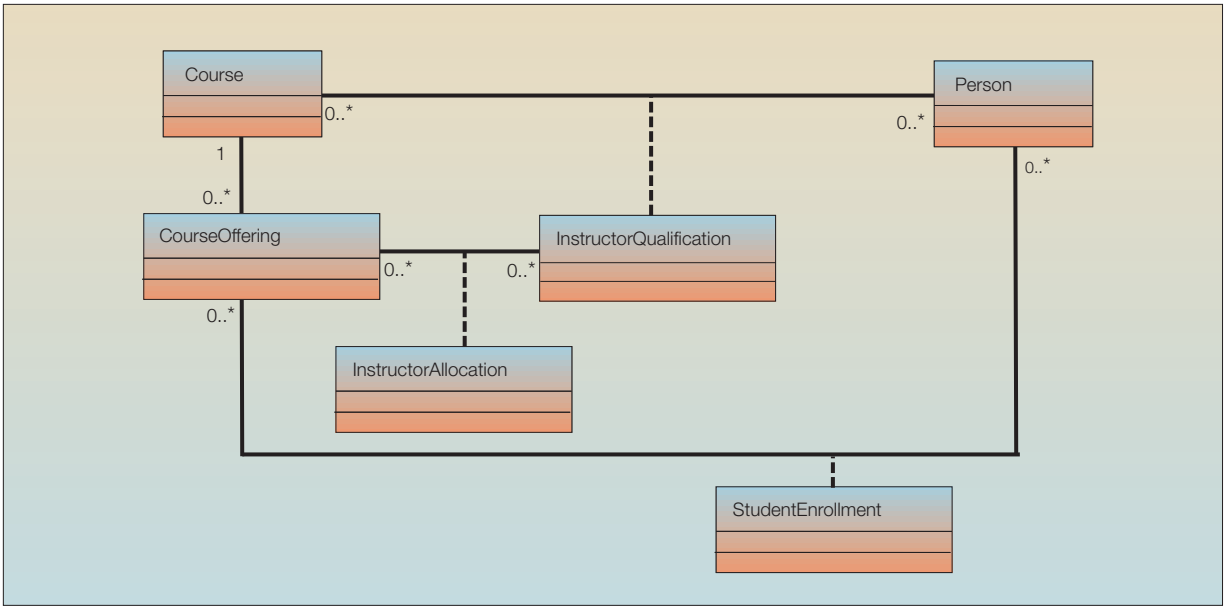
Shared persistent business objects. Shared persistent business objects represent the data and behavior of the fundamental things, both physical and abstract, that the business must manage. They are global so that they represent the requirements of the business as a whole, not the view of some user group. They continue to exist regardless of whether the application is active or not. A shared persistent business object may be part of only one database.

Shared persistent relationship business objects. Shared persistent relationship business objects are shared persistent business objects that hold the data and behavior of a relationship between two shared persistent business objects. They can be used to navigate from one shared persistent business object to another.

Property. A property is some characteristic of a shared persistent business object. A property has no meaning independent of its object.

Data type. A data type defines the nature of a property, for example, integer, Boolean, text, etc.

Figure 3 UML model of example application



Method. A method is a function specific to a shared persistent business object. A method cannot interact with a user via input/output functions; it can only interact with the external world via its parameter list.

In the example specification, the database is specified as three lists: a list of the shared persistent business objects, a list of the transactions, and a list of the events. Individual persistent shared business objects are specified in terms of their parents, their properties, and their methods. Two shared persistent business object specifications from the example specification are shown (Tables 1 and 2). Two further constructs that are used to specify the database and its behavior are now defined.

Transaction. A transaction is a function that transforms the database of which it is a component from one valid, logically consistent state to another valid, logically consistent state with no intermediate valid, logically consistent state. In other words a transaction is a single LUW. A special case is a read-only transaction. Transactions encapsulate a database, and are the only way that external components can access or update a database. A database can be considered as a large-grain object, and transactions are the methods that encapsulate it. A transaction cannot interact with a user; it can interact with the external world only via its parameter list.

Table 1 Course offering

CourseOffering	**SharedPersistentBusinessObject (a particular offering of a course, on given dates, with an instructor and a set of students)**
Parent	Course (without a course, a course offering could not exist)
Properties	courseMnemonic: Text (a short text string that uniquely identifies a course) courseNumber: Integer (a number to uniquely identify the particular offering) courseOfferingStartDate: Date status: Code (scheduled, canceled, completed) classroom: Text
Public methods	courseName( ): Text (returns the name of the related course) students( ): List (returns the list of student enrollments) instructorName( ): Text (returns the family name of the instructor, derived from person, navigated to via instructor allocation and instructor qualification)

Table 2 Student enrollment

StudentEnrollment	SharedPersistentRelationshipBusinessObject (the enrollment of a person as a student for a course offering)
Parents	Person CourseOffering
Properties	personId: Identifier courseMnemonic: Text (a short text string that uniquely identifies a course) courseNumber: Integer (a number to uniquely identify the particular offering) dateOfEnrollment: Date enrolledBy: Identifier status: Code (enrolled, waitListed, materialsSent, canceled, canceledAndInformed, completed) gradeAchieved: Code (A, B, C, D, E, F)
Public methods	studentFirstName( ): Text (returns the first name of the person enrolled) studentFamilyName( ): Text (returns the family name of the person enrolled) studentPhoneNo( ): Text (returns the phone number of the person enrolled) studentEmailAddress( ): Text (returns the e-mail address of the person enrolled) courseOfferingStartDate( ): Date (returns the start date of the course offering) courseName( ): Text (returns the name of the related course) courseMnemonic( ): Text (returns the mnemonic of the related course)

Table 3 Search course offering

SearchCourseOffering	Transaction
Input parameters	courseMnemonic, courseNumber, date, status
Returns	A set of course offerings (possibly an empty set)
Processing	Search for exact match on all parameters, but with null parameters ignored; date matches if course offering start date is equal to or greater than search date

Event. An event is a record of a significant database state change that may trigger a workflow procedure. Events carry data that are available in the context of the workflow procedure. An event is always due to the execution of a transaction, but a transaction execution need not necessarily result in an event.

Table 4 Reschedule course offering

RescheduleCourseOffering	Transaction
Input parameters	courseOffering, newDate
Returns	True or false, error message list
Processing	Set courseOffering start date to newDate; read all associated student enrollments and set status to canceled; check if new date causes instructor allocation conflict and if so cancel instructor allocation
Error messages	"Course offering not in scheduled status," "New date the same as existing date," "Technical problem—please contact the help desk quoting code xxxx"

Table 5 Course offering rescheduled

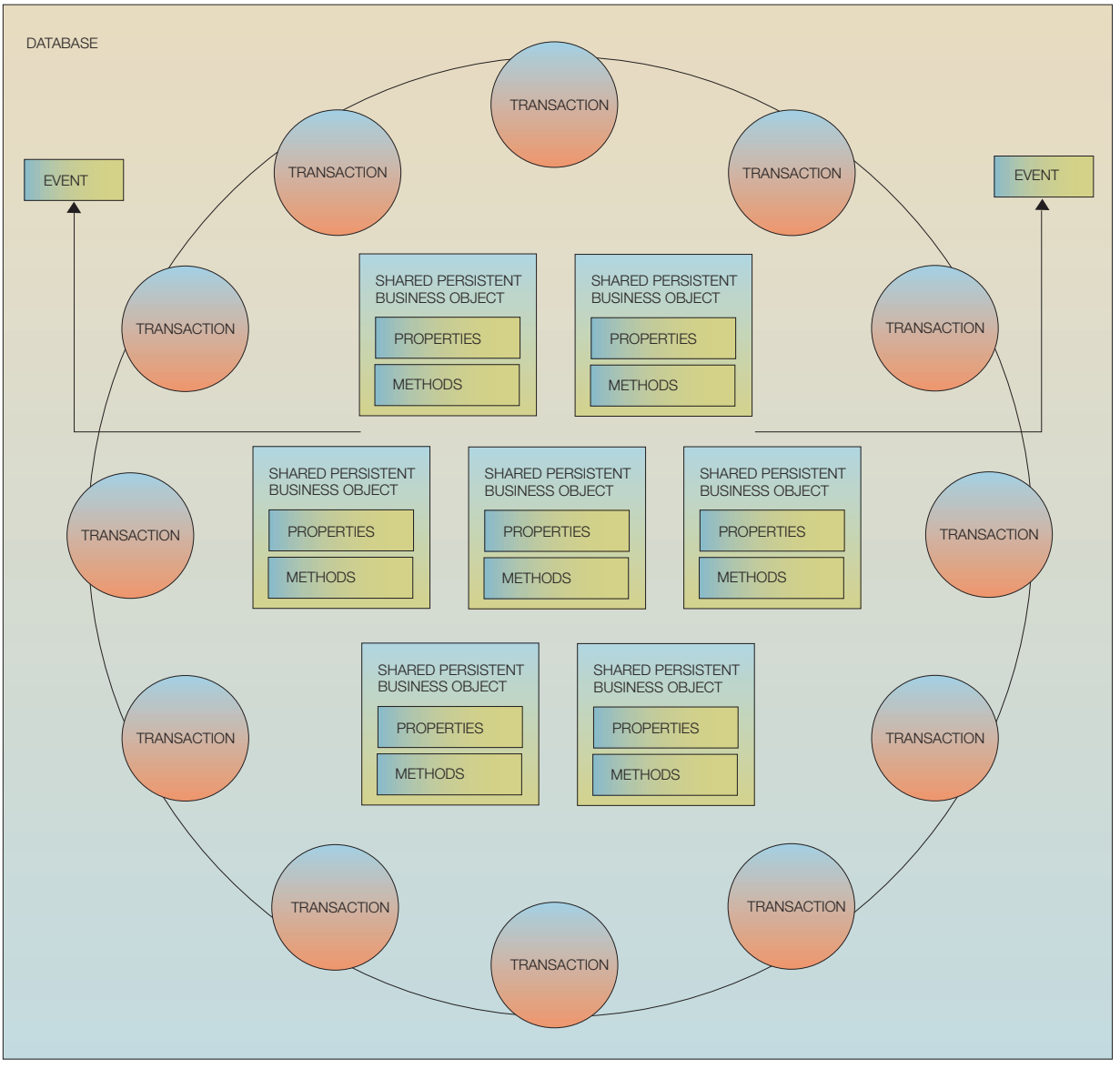
CourseOfferingRescheduled	Event (generated when the date of a course offering is changed)
Database state change	New course offering start date not equal to old course offering start date
Event data	courseOffering, oldCourseOfferingStartDate, newCourseOfferingStartDate

In the example application about 20 transactions are specified. These fall into three groups as follows:

- Search transactions, which take in a list of search arguments and return a set of shared persistent business objects
- Simple transactions that add or amend a single shared persistent business object
- Complex transactions that read and update a number of shared persistent business objects

Two transaction specifications from the example application specification are shown (Tables 3 and 4). For each transaction the specification defines the transaction name, the input and output parameter lists, the business logic that the transaction must implement, and logical error messages that the transaction may return.

Figure 4 Encapsulation of objects in the database



The business logic could be specified precisely using a formal language, or as is the case here, with a reasonably explicit description.

An example of an event specification is shown in Table 5. Events are specified in terms of a state change of the database and the data associated with the event. These data may be from either the old state of the database or the new state.

The relationships between these constructs are illustrated in Figure 4. The form of the diagram is intended to illustrate that a database is composed of a set of persistent business objects and transactions, with the transactions encapsulating the persistent business objects. The fact that the circles representing the transactions just break the boundary of the larger circle representing the database is intended to illustrate that information enters or leaves the da-

Table 6 Search course

SearchCourse	Dialog (assists the user to locate a particular course)
Input parameters	None
Returns	Single instance of course
Processing	Request search arguments from user; invoke the SearchCourse transaction; present results to user; allow user to select one course or update search arguments and retry

tabase only by the parameter lists and returned data of transactions. The only exception is that events carry data, from the database, that is available to the workflow procedures that the events trigger.

Process constructs. The constructs that abstract the various different process types are defined here. All of these processes can be considered functions in that they have a statically defined input parameter list, some processing, and the return of some statically defined data.

Note that transaction and method were defined previously, as part of the definition of constructs related to shared persistent business objects.

Function. A function is a named process that takes an input parameter list, performs some processing, and returns some data to the context from which the data were invoked. A function will have some internal variables and logic.

Dialog. A dialog is a function that, from a defined start point, reaches some desired business goal by interacting with a single user and invoking one or more transactions. A dialog cannot access the database directly; it can only access or update the database via transactions. Ideally a dialog is completed in a single session, but it may be suspended after any transaction and resumed later. A dialog may also be abandoned before it is complete. This may leave the database in an undesirable state, but not in an invalid state. (Every transaction always leaves the database in a valid state.) Dialogs must always be started by a user, either directly or via the execution of a desktop object method.

Workflow procedure. A workflow procedure is a function that, from a defined start point, reaches some desired business goal by interacting with multiple users and invoking one or more component transactions or dialogs. A workflow procedure may contain nested workflow procedures. In addition to internal logic specifying the execution sequence of component dialogs and transactions, a workflow procedure contains routing rules specifying which user will execute the next component of the workflow procedure. Like a dialog, a workflow procedure can only access or update the database via transactions. Unlike a dialog, a workflow procedure may be initiated by an event as well as by a user, and its internal flow logic may also be controlled by events.

Workflow procedures, dialogs, methods, and functions can be nested; transactions cannot. Note that each transaction will normally have a matching dialog that allows the user to enter the transaction parameter values, review any error messages returned by the transaction, retry the transaction if necessary, and finally receive confirmation that the transaction has executed successfully. In some cases a single dialog will provide these facilities for several transactions.

Dialogs are functions and have an input parameter list. The prime use of this parameter list is to allow desktop object methods that invoke dialogs to place restrictions on the parameter lists of the transactions invoked by the dialogs. This allows a desktop object method to provide restricted function without the need for a special dialog and transaction.

Both dialogs and workflow procedures have a range of characteristics that are best expressed as categorizations of individual instances rather than as subclasses. For instance, both dialogs and workflow procedures are functions, and as such have internal logic as well as the ability to invoke component transactions. This logic may interact with a user to determine the internal flow. At one extreme the internal flow may be totally fixed—the user can enter data but cannot affect the execution sequence of component transactions. At the other extreme a dialog or workflow procedure may allow the user complete control over which transactions are executed and when they are executed. Most real dialogs and workflow procedures lie somewhere between these two extremes.

Specifications of two dialogs from the example application are shown (Tables 6 and 7). The form of the specification is similar to the transaction specification in that it specifies the dialog name, input parameters, return value, and processing.

For a simple application it is difficult to provide realistic examples of workflow procedures, so the example shown here is minimal. It defines the procedure to be followed when a course offering is rescheduled. In the education center, courses can only be rescheduled by the center administrator, who uses the "reschedule course" transaction to reset the course offering start date and set the status for all student enrollments to "canceled." The procedure that the education center follows in this case is for a course secretary to phone each student. The course secretary will determine whether the student can accept the new start date and will either set the student status to "enrolled" or "canceled and informed." Since multiple users are involved, a workflow procedure is required. In fact, the workflow procedure itself contains only a single dialog executed by a single user, but because the original cancellation transaction and the handling of the results are done by different users, the routing rules of a workflow procedure are required.

Text is clearly not a very good way to express workflow procedures. If workflow procedures are going to be part of the application, then the normal practice would be to use the workflow procedure definition facilities provided by the workflow manager. One of the example workflow procedure specifications is shown in Table 8. It is similar to the other processing specifications, the main difference being the specification of the event that triggers the workflow procedure.

The relationships and defining characteristics of the various process types are illustrated in Figure 5.

User interface constructs. The user interface defined by the constructs is inherently multithreaded and may be either object/action, action/object, or a mixture of both.[5] It is defined in terms of desktop objects and the dialogs and workflow procedures that the user can invoke. The desktop objects have properties that the user may view, and methods that the user may execute. Each desktop object is derived from a single root, a shared persistent business object. Their properties may be properties of the root shared persistent business object, the results of methods of the root shared persistent business object, which may comprise data extracted from other related shared persistent business objects, or contained desktop objects whose roots are derived from the root shared persistent business object.

Table 7 Handle course offering rescheduling (dialog)

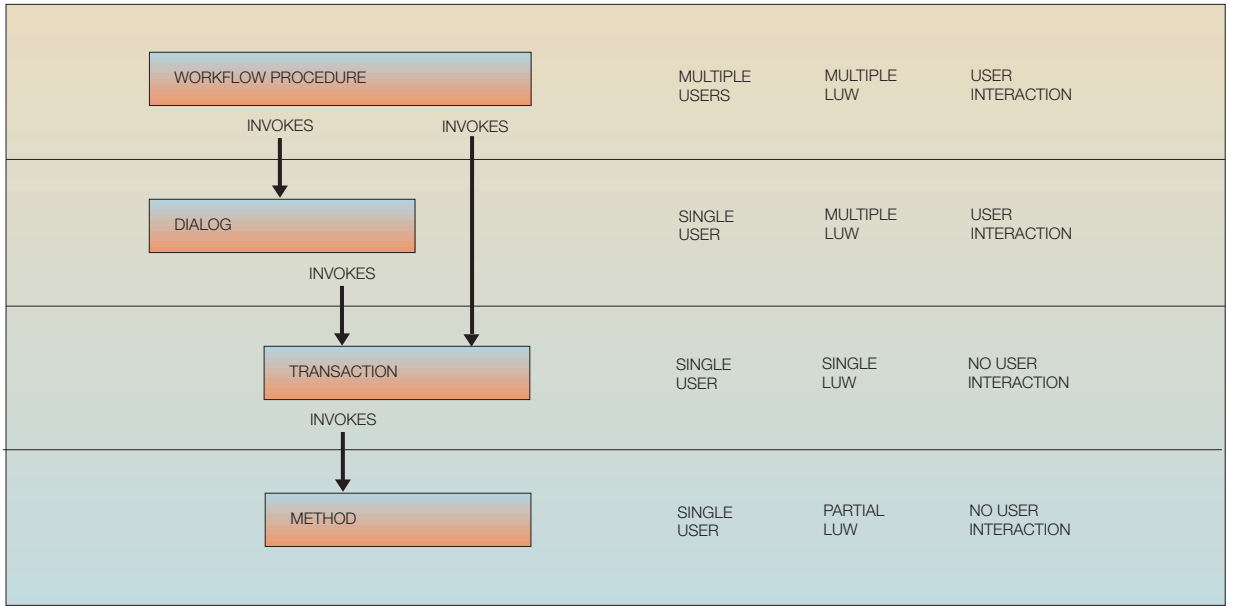
HandleCourse-Offering-Rescheduling	**Dialog (prompts the user to call each student from a rescheduled course offering and determine whether the student will accept the new date)**
Input parameters	courseOffering, oldCourseOfferingStartDate
Returns	Nothing
Processing	For each student enrollment: Display user prompt window giving student and courseOffering details (including oldCourseOfferingStartDate and controls to record call not made)—student will accept new date or wish to cancel. If student will accept new date, issue an AmendStudentEnrollment transaction to set status to "enrolled" without any further user interaction. If student will not accept new date, issue an AmendStudentEnrollment transaction to set status to "canceled and informed" without any further user interaction.
Error messages	Present transaction error messages to the user

Table 8 Handle course offering rescheduling (workflow procedure)

HandleCourse-Offering-Rescheduling	**WorkflowProcedure (contact students when course offering is rescheduled)**
Triggering event	courseOfferingRescheduled
Input parameters	courseOffering, oldCourseOfferingStartDate (from event)
Returns	Nothing
Procedure flow	Invoke the CourseOfferingRescheduled dialog for any user in the course secretary user group

The desktop object methods are simple mapping functions that invoke transactions, dialogs, or workflow procedures, optionally supplying some or all of the input parameters from the desktop object properties. Desktop objects provide a package of data and function that is designed to improve the usability of

Figure 5 Relationships and characteristics of process constructs



the application for a particular group of users. The constructs are defined here.

Application. An application is a collection of desktops defined for management and control purposes.

Desktop. A desktop is an object that allows a user to interact with certain application components contained within it. It may contain other desktops, desktop objects, dialogs, or workflow procedures. A user may instantiate desktop objects to populate the desktop, and may then execute their methods. A user may also directly invoke and interact with any of the dialogs or workflow procedures contained in the desktop.

Desktop object. A desktop object is a collection of desktop object properties, derived from a single root shared persistent business object, and a set of desktop object methods. It provides a particular bundle of data and function that is designed to make interaction with the system easier for some group of users. A user may view the desktop object properties or invoke any of the desktop object methods. A desktop object may also contain other desktop objects as properties.

Desktop object property. A desktop object property is any property of the root shared persistent business object, or the result of the execution of any read-only method of the root shared persistent business object. Use of read-only method results allows a desktop object to contain data derived from any shared persistent business object that can be navigated to from the root object. It also allows a desktop object property to be a complex data structure.

Desktop object method. A desktop object method is a function that invokes one or more transactions, dialogs, or workflow procedures. If all the parameters required by any invoked transaction or workflow procedure can be supplied from the desktop object data, then they may be invoked directly. If further user input is required to complete a parameter list, then the desktop object method must invoke a dialog that will provide the necessary user interaction. The dialog will then invoke the transaction or workflow procedure. The set of desktop object methods can be viewed as a drop-down menu list associated with the desktop object.

Note that no windows are defined explicitly. Windows are implicitly defined by the desktop and its contents, the properties of desktop objects, the function signatures of transactions, dialogs, and workflow procedures, and I/O functions invoked by dialogs to communicate with the user. The graphics and layout of windows are part of the implementation

Table 9 Center administrator desktop

CenterAdministratorDesktop	Desktop
Properties	courses: [CourseDesktopObject] /* represents an unsequenced collection of course desktop objects */ courseOfferings: [CourseOfferingDesktopObject] instructors: [InstructorDesktopObject] students: [StudentDesktopObject] addCourseDialogs: [AddCourseDialog] addCourseOfferingDialogs: [AddCourseOfferingDialog] addPersonDialogs: [AddPersonDialog]
Methods	addCourse (invoke the AddCourse dialog and from the returned course instantiate a CourseDesktopObject) retrieveCourse (invoke the CourseSearch dialog, and from the returned course instantiate a CourseDesktopObject) addCourseOffering (invoke the AddCourseOffering dialog, and from the returned course offering instantiate a CourseOfferingDesktopObject) retrieveCourseOffering (invoke the CourseOfferingSearch dialog, and from the returned course offering instantiate a CourseOfferingDesktop object) addStudent (invoke the AddPerson dialog and from the returned person instantiate a StudentDesktopObject) retrieveStudent (invoke the StudentSearch dialog, and from the returned person instantiate a StudentDesktopObject) addInstructor (invoke the AddInstructor dialog and from the returned person instantiate an InstructorDesktopObject) retrieveInstructor (invoke the InstructorSearch dialog, and from the returned person instantiate an InstructorDesktopObject)

Table 10 Course secretary desktop

CourseSecretaryDesktop	Desktop
Properties	courses: [CourseViewDesktopObject] courseOfferings: [CourseOfferingViewDesktopObject] instructors: [InstructorViewDesktopObject] students: [StudentDesktopObject] addPersonDialogs: [AddPersonDialog] addStudentEnrollmentDialogs: [AddStudentEnrollmentDialog]
Methods	retrieveCourseView (invoke the CourseSearchDialog and from the returned course instantiate a CourseViewDesktopObject) retrieveCourseOfferingView (invoke the CourseOfferingSearchDialog and from the returned course offering instantiate a CourseOfferingViewDesktopObject) AddStudent (invoke the AddPersonDialog and from the returned person instantiate a StudentDesktopObject) retrieveStudent (invoke the StudentSearchDialog and from the returned person instantiate a StudentDesktopObject) retrieveInstructorView (invoke the InstructorSearchDialog and from the returned person instantiate a CourseViewDesktopObject)

design, and are deliberately absent from the business function design.

The example application has two desktops. One is designed for the education center administrator. It provides access to all the function of the application, and in particular allows courses, course offerings, instructor qualifications, and instructor allocations to be added. The second is designed for a course secretary. This allows all the application data to be displayed, but only allows for persons and student enrollments to be added.

Table 11 Desktop objects in the example application

Course	Derived from Course persistent business object and contains CourseOffering and InstructorQualification desktop objects
CourseOffering	Derived from CourseOffering persistent business object and contains course data, instructor name, and StudentEnrollment desktop objects
StudentEnrollment	Derived from StudentEnrollment persistent business object and contains data from Person, CourseOffering, and Course persistent business objects
Student	Derived from Person persistent business object (with no instructor qualification for this course), and contains person data plus StudentEnrollment desktop object
Instructor	Derived from Person persistent business object with instructor qualification for this course, and contains person data plus InstructorQualification and InstructorAllocation desktop objects
InstructorQualification	Derived from InstructorQualification persistent business object and also contains data from Person and Course persistent business objects
InstructorAllocation	Derived from InstructorAllocation persistent business object and also contains data from Person, Course, and CourseOffering persistent business objects

Table 12 Course offering desktop object

CourseOfferingDesktopObject	**DesktopObject**
Root shared persistent business object	CourseOffering
Properties	All properties of CourseOffering persistent business object, plus courseName ( ) instructorName ( ) list of StudentEnrollment desktop objects, for which roots are provided by students( ) method of CourseOffering persistent business object
Methods	amend (allows certain course offering properties to be directly amended by the user, though in this simple example the only property that can be directly modified is classroom) enrollStudent (invokes the AddStudentEnrollment dialog, supplying the courseOffering parameter from desktop object data. From the returned StudentEnrollment shared persistent business object it instantiates a StudentEnrollment desktop object, and adds it to its set of StudentEnrollment desktop objects) allocateInstructor (invokes the AddInstructorAllocation dialog) rescheduleCourseOffering (invokes the AmendCourseOffering dialog, which will issue a RescheduleCourseOffering transaction, which will generate a CourseOfferingRescheduled event, which in turn will trigger the HandleCourseOfferingRescheduling dialog) cancelCourseOffering (invokes the CancelCourseOffering dialog, which will issue a CancelCourseOffering transaction, which will generate a CourseOfferingCanceled event, which in turn will trigger the HandleCourseOfferingRescheduling dialog) recordCourseOfferingCompletion (invokes the CourseOfferingCompletion dialog)

The desktops are themselves objects, and their classes are specified in terms of their data and methods. The desktop data are collections of desktop objects, dialogs, and workflow procedures. Desktop methods allow the user to invoke the dialogs and workflow procedures. These methods are trivial and can be assumed from the presence of the dialogs and workflow procedures in the desktop data. Desktop methods also allow the user to add desktop objects to the desktop, giving the user access to the desktop objects' data and methods. These methods are not trivial, so they are specified explicitly. For the exam-

ple application, the specification of the center administrator desktop is shown in Table 9 and the specification of the course secretary desktop is shown in Table 10. The desktop objects in the example application are shown in Table 11.

In addition there are "view" versions of Course, CourseOffering, and Instructor desktop objects. These have the same data as the normal versions but do not have the methods that allow the underlying persistent shared business objects to be updated.

Desktop objects can contain lists of other desktop objects, so they provide a hierarchical, denormalized view of the persistent shared business objects. Via their methods they give access to the application workflow procedures, dialogs, and transactions. They can appear on screen in a number of different visual forms. One is as an expandable indented tree structure, with pop-up windows for individual object details, as used for viewing file directory structures. Another is a "forms" style with a header section and with scroll boxes for the next level down in the hierarchy. Multiple scroll boxes can be used to cover multiple hierarchical levels. Desktop objects are specified in terms of their properties and methods as shown in Table 12.

The major relationships between the user interface constructs and the key related constructs are illustrated in Figure 6. The arrows indicate that execution of a desktop object method will invoke a transaction, dialog, or workflow procedure.

Dialogs are logically part of the desktop—dialogs interact with users, and the desktop is defined as the object that contains the application's components, with which the user may interact. Instances of the dialogs are particular executions, and the desktop allows for multiple instances to be available at the same time. Dialogs may take some period of time to complete, so it is important to have multiple instances available. This supports the ability to suspend the execution of one dialog, due to an interruption, and start the execution of another dialog of the same type. Once a dialog has completed execution, the instance in the desktop provides a client-side log. How long completed dialog instances are kept in the desktop is an implementation design choice. They may be discarded immediately, or kept for some period of time.

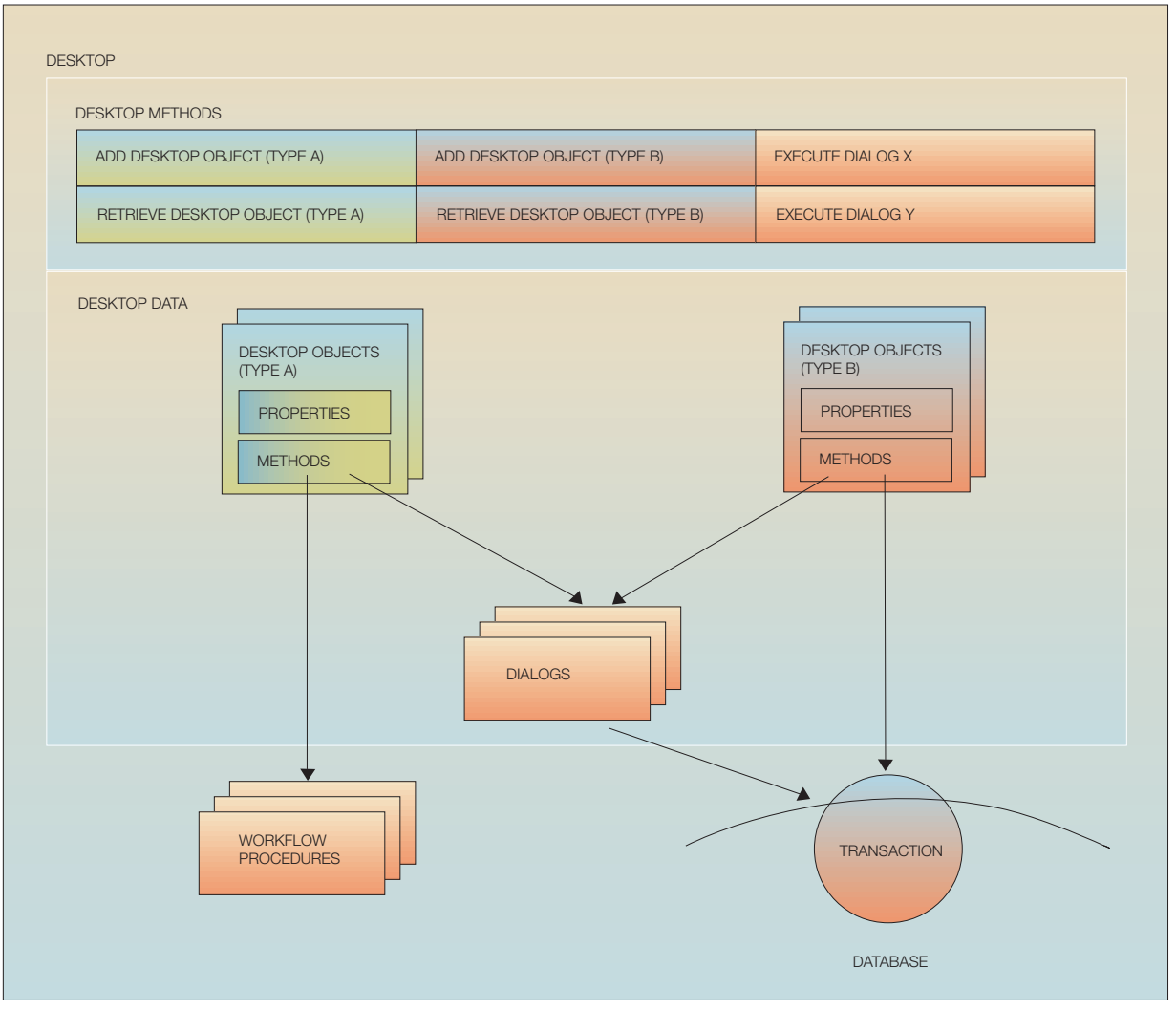
Logical application structure

The constructs proposed give rise to a logical application structure that can be represented diagramatically as shown in Figure 7. It is a highly layered structure. A set of persistent shared business objects and their methods provide the basic components. Transactions are built from methods and functions. Databases are large-grained objects built from the shared persistent business objects and their transactions. Dialogs are built from transactions and functions. Workflow procedures are built from other workflow procedures, dialogs, transactions, and functions. Desktop objects are built from data defined from existing persistent object properties and methods, and from desktop methods that are mapping functions onto already-defined transactions, dialogs, and procedures. Desktops are composed of already-defined dialogs, procedures, and desktop objects. This highly layered structure maximizes reuse during the analysis activity. It also maximizes the flexibility of the implemented system. It allows rapid implementation of dialogs and workflow procedures built from existing transactions. User interfaces can also be assembled rapidly using already-defined shared persistent business objects, transactions, dialogs, and workflow procedures.

Within the constructs there is no inherent duplication of business logic. Business logic is expressed only once. This minimizes analysis effort, and eliminates possible errors and ambiguity. This absence of duplication at the analysis level does not imply that there is no duplication of business logic in the implemented application. There are sound reasons for duplicating some business logic in the implemented application. These reasons are covered later.

There is, however, duplication of data in that the desktop objects contain copies of data derived from the persistent shared business objects. This duplication provides a hierarchical view of the underlying persistent data contained in the database. The database will normally be in third normal form[6] and the desktop objects are a denormalization designed to improve usability for a particular user group. Multiple desktops may be defined, and each may contain a different set of overlapping desktop objects, so multiple denormalizations are possible, each targeted at a different user group. Note that these denormalizations are quite different from the single denormalization that will be part of the implementation of the shared persistent business objects on a particular database platform. The platform implemen-

Figure 6 Desktop methods in user interface invoke high-level process constructs.



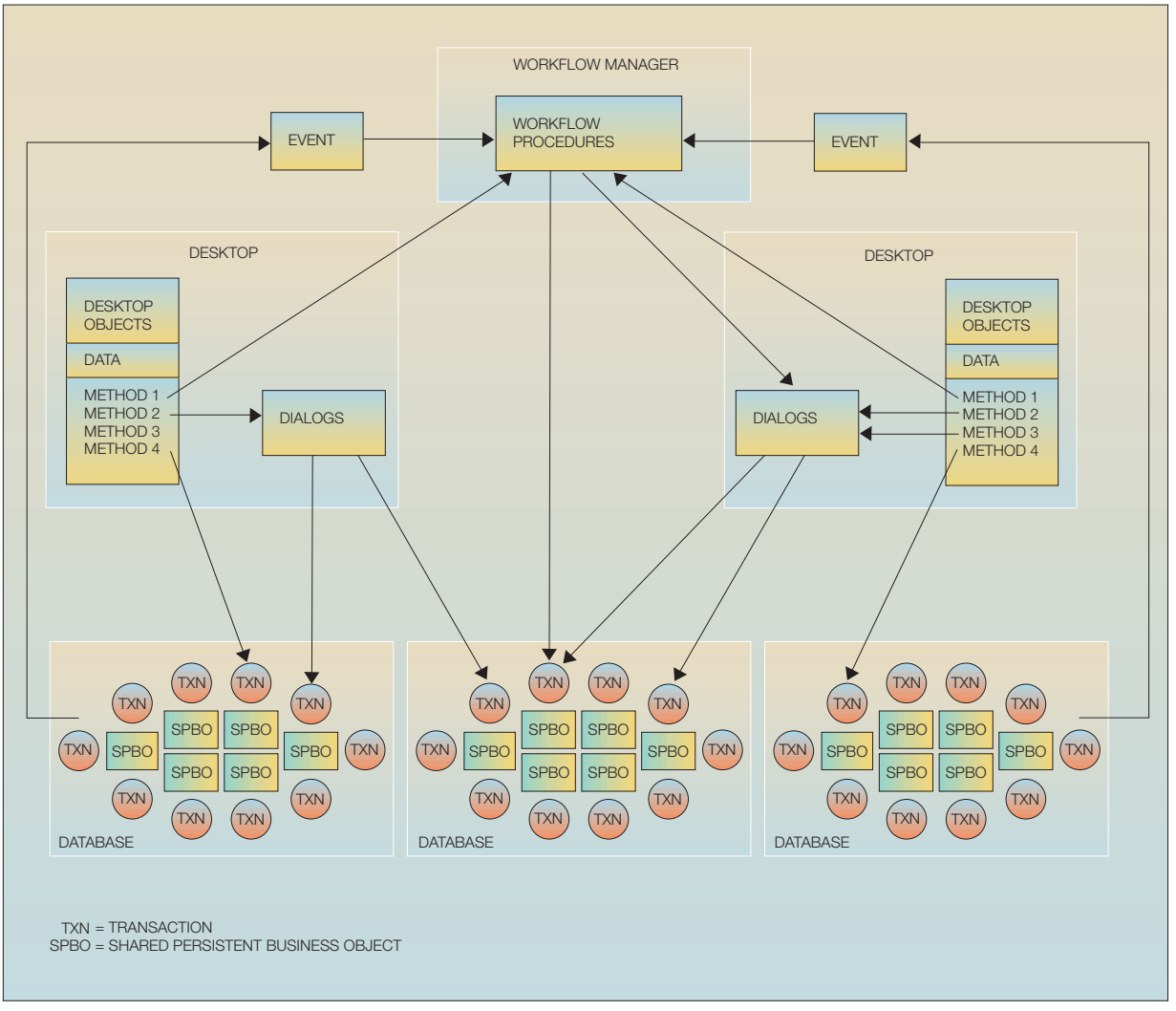
tation denormalization is a single global denormalization, primarily aimed at providing acceptable database performance.

Analysis activity

The constructs proposed do not imply any one application development method. While the deliverables from analysis are consistently defined, there are various possible methods for producing them. Some application development methods are more focused on process decomposition, while others are data- or object-based. Some application development methods are based on very short iterative development cycles, while others use a more traditional "waterfall" approach. Whatever method is used, the analysis constructs proposed can guide the analysis activity of commercial applications, and can provide deliverables that are more precise and thus provide better input to subsequent development activities.

Use of the constructs will also ensure that logical units of work are defined as an inherent part of the analysis, and not as a separate additional activity.

Figure 7 Logical structure of proposed constructs



Neither traditional structured analysis[7] involving top-down process decomposition, nor the use-case-driven approach[8] normally identifies logical units of work as an inherent part of the business analysis activity. In commercial applications the logical units of work are central, and their identification should therefore form an inherent part of the analysis activity.

Use cases can be useful in commercial applications for the initial unstructured capture of user requirements for business processes. However, they must be further decomposed to explicitly identify the workflow procedures, dialogs, and transactions involved. Top-down process decomposition has been bedeviled by the lack of any sound definition of the various subclasses of process. While "manage IBM" and "update quantity in stock" were clearly rather different kinds of process, it was never clear what intermediate kinds of process were required through the decomposition process, and how these should be defined. The constructs proposed provide a semantically sound set of process subclasses based on logical units of work. These enable process decomposition to be a well-defined, well-structured activity, which directly produces deliverables that can be mapped to an implementation platform in a straightforward way.

Similarly the user interface constructs allow a user interface to be designed at a logical level. The data that a user can view and the processes that can be invoked are well-defined, while details of screen design and layout are left to the platform design stage.

Relationship to business modeling

These constructs are proposed for application specification. Application specification is quite distinct from business modeling and business process re-engineering, which seek to understand and redesign fundamental business processes without regard to the automation of information processing. The recent paper by McDavid[9] provides an approach to the description of fundamental business architectures, and the book by Scheer[10] provides a current approach to business modeling. Business modeling and re-engineering should precede any application specification and should be followed by definition of the scope of any applications that automate part or all of the information processing involved. Once an application's scope has been defined, its detailed business function can then be specified using the constructs proposed here.

Communication with business users

The proposed constructs are designed to improve communication between business analysts and implementers. However, the obvious question arises as to what extent the specification of an application using these constructs can be used to communicate its function back to business users. Some further help beyond the application specification delivered to implementers is certainly required. This could be supplementary documentation and informal diagrams, specifically targeted at business users. The author's preference would be to develop a prototype that is the simplest possible single-machine implementation of the application, effectively an executable form of the business function specification. This would allow users to exercise and verify the business function specified. It would also facilitate the development of test cases for the full implementation of the application.

Mapping to a platform

The constructs provide a template for structuring the deliverables from analysis. This makes the task of the business analysts easier, but more importantly it also makes the task of the platform designers much easier too. Instead of a large number of undifferentiated analysis classes, and use cases that do not address the fundamental business issues surrounding units of work, platform designers are presented with well-structured input that has addressed the key issues and can be mapped to a suitable platform in a straightforward way.

The key point about the platform mapping is that it does not depend on the particular business function of the application. The mapping depends on the construct being mapped, and nonfunctional requirements, such as performance. So a transaction will map to a platform in a particular way because it is a transaction, and because it has a particular response requirement, a particular pattern of database accesses, a particular volume, etc. Whether the business function is general ledger or order entry is completely irrelevant. This means that much platform design can be done at a generic level, in parallel with business analysis. Standard mappings can be defined and reused depending on the nonfunctional requirements of each construct instance.

In general it is desirable to map the analysis constructs onto a platform in the simplest possible way, so that as much as possible there is a one-to-one correspondence between analysis constructs and classes in the implementation. The constructs have been designed to map easily to a client/server platform. The shared persistent business objects and associated transactions naturally map to an object database with proper concurrency control, or a transaction manager and relational database such as CICS*/DB2* (Customer Information Control System/DATABASE 2), or an object request broker using a relational database manager to provide persistency. The dialogs and user interface constructs naturally map onto an object-oriented client. If workflow procedures are defined, then a global workflow server is also required. Particular platforms will often require somewhat more complex mappings, which are outside the scope of this technical note. There are also some platforms onto which the full set of constructs cannot easily be mapped, because these platforms lack capabilities that are inherent in the constructs. For instance, a mainframe with "dumb" terminals lacks the ability to support a multiwindow user interface, which is inherent in the constructs proposed.

Whatever the platform, there are a number of areas where platform designers will usually need to significantly transform the analysis model defined by the constructs. These are described here.

Database design. The analysis model database, defined by the set of shared persistent business objects, should be in third-normal form in terms of the business domain that it covers. This maximizes the inherent flexibility of the system and avoids duplication of data. However, with current technology, a straightforward implementation leads to unacceptable performance. It can also make the coding of database accesses unnecessarily complex. Appropriate physical database design is required to ensure adequate performance. This is a well-established discipline[6] and the constructs do not require anything new.

As mentioned earlier, two rather different denormalizations are required. One is the denormalization to arrive at the physical database design. This is a single denormalization that is part of the platform mapping and is driven by the requirements across all users. It represents a global compromise. The other denormalization is the denormalization represented by the desktop objects, which is part of the business function design. Here the denormalization is targeted at a particular group of users, and many different and overlapping denormalizations are possible. These denormalizations are concerned with ease of use, not with performance. One of the problems with some application architectures is that they provide only a single denormalization to meet these two quite different requirements.

Logical unit of work consolidation. The definition of a transaction, and thus of an LUW, is very precise. It needs to be in order to provide a solid foundation for all the other constructs that are built around it. However, it will tend to result in rather fine-grained transactions. This may give performance problems, caused by heavy network traffic and many small database updates. Fine-grained transactions can also affect usability by forcing dialogs to involve many small transactions, rather than a few transactions with larger scope. Another factor is that a business may make a policy decision to combine two LUWs, even though the analysis has shown that they can be separated. For instance, adding a person object and adding the link to a previously known address are logically two separate LUWs. However the business may make a policy decision not to keep personal details unless an address is also recorded. It may therefore decide to roll up the two LUWs into a single LUW.

So the implementation may well choose to combine some LUWs. There are two different mechanisms that can be used to provide this LUW consolidation. The first is to define transactions that combine multiple logical LUWs into a single unit of work in the implementation. This need not be on an exclusive basis. For instance, in the previous example two transactions could be provided, one to add a person and an address link and one to add just an address link. For a new person the combined transaction is used, but for address updates the add link transaction would be used. Proper management of the implementation should, of course, ensure that source code is reused and not duplicated.

The second alternative is to define a dialog that makes use of the facilities provided in transaction managers—to widen the unit of work to span more than one transaction. Note that this approach implements business policy decisions to widen the unit of work, but will not have the same impact on performance as combining transactions.

Business logic duplication. Dialogs, as defined earlier, need contain no business logic other than that required to ensure the necessary transactions are executed in the correct sequence. If a user enters invalid data for a transaction, then transaction error messages can be shown to the user, who can make the necessary corrections. However, this is not particularly user-friendly, and it requires additional server interactions, which cost time and resources. Because dialogs are simply client-side programs that execute database updates via external calls, they may contain business logic that validates the data entered before transactions are sent to the server. In this way validation can be immediate, and load on the network and the server is reduced. In the extreme this approach can involve a highly sophisticated client-side dialog that accumulates user input in a scratchpad area. Then when all validation is complete and the user is satisfied with the input, a number of server transactions are invoked without further user interaction. As transactions must contain all the validation logic necessary to guarantee the integrity of the database, any client-side validation logic will be a duplication of logic in the transactions. There is a trade-off here between usability and performance, on the one hand, and the costs of writing and maintaining duplicate business logic, on the other hand. The implementation design will normally be skewed toward usability and performance for high-volume business-critical dialogs, and toward cost saving for low-volume dialogs.

Window combination. Separate windows are implicitly defined by the desktop and its contents, the prop-

erties of desktop objects, and the dialogs with their user interaction. A pull-down list is also implicitly defined by the set of desktop object methods.

To improve usability, the user interface designers may choose to combine some of these logical windows into a single window. For instance, a desktop object properties window will usually be combined with an "amend" dialog window. The combined window will display the object properties, and the logical amend dialog window will be provided by allowing changes to some of the displayed properties, along with an error message area and an "amend" button.

Note that these design transformations do not represent a deficiency of the analysis constructs. To provide a proper separation of concerns and a better understanding of what is fundamental, the constructs abstract and separate into a layered structure things that are logically separate. However, the real world in which the implementation must exist is more complex, with many additional and conflicting requirements that are deliberately not considered during business analysis. Platform implementation must take these additional requirements into consideration and therefore must further transform the analysis model.

Internet platforms

Initially Web sites were purely information providers and therefore not commercial applications as defined here. Now there is a very strong move to incorporate user interfaces to commercial applications into Web sites. The question arises as to whether an Internet-based implementation of a user interface is just another platform implementation, with its own particular technical considerations, or whether there are some different aspects of an Internet implementation that need to appear in the business function specification. The key difference of most commercial application desktops that are part of a Web site is not that they use Internet technology, but that they are designed for a user who is a customer or a supplier, not an employee. This means that users are only allowed to manipulate data that relate to them, for instance their bank account or their orders. So the business function specification of a commercial application with an Internet-based customer interface will usually need to include an additional desktop with some slightly different desktop objects from the normal employee desktop. The desktop objects will be largely the same, and the main difference will be that the desktop methods to add desktop objects will only allow the creation and retrieval of instances that are directly related to the user.

Origins and context

The constructs proposed here draw on a number of sources. The first is the early work on relational database theory,[11] and experience of the implementation of relational databases.[12] This led to business analysts defining entity-relationship logical data models in third normal form. Implementers transformed these into denormalized physical database designs,[6] taking into account nonfunctional requirements and the specifics of the database software. This aspect of commercial application development has worked well for some time. What is proposed here for persistent data supports a widely used object-oriented form of these ideas, as expressed for instance by the "specification" static object model defined by Cook and Daniels.[13]

A key source for the process constructs was the experience in the early 1990s of developing major new customer service systems for a number of utilities in the United Kingdom. These client/server systems consisted of a relational database server and a client that provided a sophisticated object-oriented user interface. A key requirement of these systems was to include workflow management. Initially the developers had a two-level mental model of processes consisting of transactions, as implemented by CICS and other similar software, and workflow procedures as implemented by the workflow managers that were then starting to appear. However, it became apparent that the case where the same person executed successive steps in a workflow procedure required quite a different implementation and was more than just a minor special case. This led to the definition of dialogs as a further important process subclass, and the recognition by the business analysts involved that the three-level structure seen in the implementation was a reflection of a three-level structure that should also exist in a logical, platform-independent process model. The process constructs proposed here provide a more tightly defined version of what was learned during this period. They form a logical process model that can be transformed into an implementation with a logical-to-physical transformation similar to that used successfully for data for a long time.

At the same time object technology, with the encapsulation of data and its close binding with process,

was beginning to influence commercial application developers. Objects seemed useful both for describing the persistent business entities with their behavior and for the implementation of graphical user interfaces. The appearance of the model-view-controller (MVC) design pattern[14] showed the value of the separation of the on-screen view from the underlying business data. It led to the idea of a logical user interface as a collection of objects where the user could invoke the object methods.

While all these ideas from different sources were useful, initially they did not seem to fit together in a coherent way. It was not clear how the notion of a method fitted in with wider processes, such as workflow procedures and dialogs that experience had shown were a necessary part of commercial applications. Although work has been done to bring workflow and objects together[15] this did not include the notion of a dialog or the user interface aspects of an application. Also it was not clear whether the model objects of the MVC pattern seen in the user interface and the relational tables that usually provide persistence were the same things at the functional specification level.

What is proposed here represents the author's attempt to bring together a range of ideas, that have proved useful in commercial application development, into a consistent set of related constructs. In particular the intention has been to extend the separation between logical design and physical implementation—which has worked well for data—to all aspects of the design of a commercial application; in other words, to provide a logical specification of the complete application. What makes this intention realistic is that only systems comprising a well-defined subclass are covered, namely commercial applications.

Current status and future directions

These proposals are published to allow scrutiny and review by the commercial application development community. They are the result of a period of reflection on the lessons learned from a number of projects, and a preparation for improving the development process on future projects. They have not yet been proved to work on an actual project. As yet there has been limited opportunity to use the constructs during application development. They have been tested by using them to specify example applications, and there has been some limited use of the process constructs on a small project. Rather than wait until more experience has accumulated, the ideas are published as a technical note so that the scrutiny and review of a wider community can more quickly determine whether what is proposed adds real value.

Besides their eventual use on a live project, there are a number of other directions in which these proposals could be further developed. In the examples, the constructs were specified using natural language with some additional formatting and layout. Clearly a precise formal language, in which all the constructs could be specified in the simplest possible form, would be of value. An open issue is whether the constructs are sufficiently well-defined that a precise formal language is possible. Many of the constructs already appear in formal languages but others are new. The author believes that all of the constructs could be specified in a precise form, possibly with some further tightening of the definitions. However, this has yet to be demonstrated.

Some work has been done in this direction. Another source of the ideas expressed here was an IBM project to develop a tool for the expression of platform-independent business rules and their compilation into code that could be dynamically incorporated into installed applications. This project developed a programming language[16] in which to specify the business rules and a compiler. The programming language combined an object-oriented language, a functional programming language, and a database query language. It was able to express precisely, in a platform-independent form, all the constructs required to specify shared persistent business objects and their behavior. The author has defined some straightforward extensions to this language that allow the specification of transactions, desktop objects, events, and most dialogs. Workflow procedures were not covered, as they require a significantly different language. Although this experimental language had some very attractive features, it would make sense to base any formal expression of the constructs on a mainstream programming language.

Another possible direction is to produce prototypes that would provide the simplest possible executable form of the application. This would have value in providing a form of the specification that can be utilized to verify that the specification meets the users' needs. It would be a valuable exercise to determine how the constructs map into code, and whether information is being added by the programmer that should be specified in the constructs. An executable

prototype of the example application has been produced using the Java** language and the Abstract Window Toolkit (AWT). It is available as a Java applet along with the .java source files.[3] It is a complete implementation of the example application with the exception of the workflow procedures and events. It maps all the constructs, one-to-one, into Java classes. The only additional classes required were those required to define the windows needed to view the desktop objects, plus some utility classes to provide parameter lists and error message handling. The prototyping exercise led to some minor refinement of the construct definitions, and also demonstrated that the constructs are complete enough to be mapped into an executable application.

Summary

The constructs proposed are intended to provide a better set of analysis deliverables for commercial applications. They enable analysts to define a platform-independent abstraction of:

- The fundamental business objects
- Extended processes involving multiple LUWs and multiple users
- The interaction of the user with the application

They provide a highly layered model that builds more complex components from simpler, more basic components, providing the maximum reuse during both analysis and implementation.

For implementers, the constructs can provide input that is organized and structured in a familiar and useful way. The mapping of the business function onto the platform can follow a number of standard patterns, based on the constructs, the nonfunctional requirements, and the choice of platform. These patterns can be designed in parallel with business analysis, leading to a shorter development cycle.

The constructs enable a proper separation of concerns between business analysts and platform designers. They deliberately exclude some aspects of the final application. What they include is the domain of the business analyst, and what they exclude is the domain of the platform implementation designer.

They do not address every problem that has affected commercial application development, but they do address the key problem of handing over, from the development team that is primarily business-focused, to the development team that is primarily technology-focused, a complete and understandable specification.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references and notes

1. The *IBM Systems Journal* **38**, No. 1, provides a number of papers on the Enterprise Solutions Structure (ESS).
2. I. R. Forman and S. H. Danforth, *Putting Metaclasses to Work*, Addison-Wesley Publishing Co., Reading, MA (1998).
3. Contact the author: david_bevington@uk.ibm.com.
4. M. Fowler with K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Publishing Co., Reading, MA (1997).
5. For an "action/object" interface an action is selected first, usually via a hierarchical set of menus that offer an increasingly specific action, and then the data are specified that the action is to be performed upon. This is the usual style for traditional "green screen" applications. For an "object/action" interface, an object is selected, then a set of possible actions that may be performed on it is made available.
6. C. S. Mullins, *DB2 Developer's Guide*, Sams Publishing, Indianapolis, IN (1997). Pages 115–126 provide a brief definition of first, second, and third normal forms, and a list of the denormalizations typically employed by physical database designers.
7. E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, Inc., Upper Saddle River, NJ (1989).
8. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Publishing Co., Reading, MA (1995).
9. D. W. McDavid, "A Standard for Business Architecture Description," *IBM Systems Journal* **38**, No. 1 (1999), pp. 12–31.
10. A.-W. Scheer, *ARIS—Business Process Frameworks*, Springer-Verlag, Berlin (1999).
11. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* **13**, No. 6 (June 1970).
12. C. J. Date, *An Introduction to Database Systems, Sixth Edition*, Addison-Wesley Publishing Co., Reading, MA (1995).
13. S. Cook and J. D. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice-Hall Canada, Scarborough, Ontario (1994).
14. G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* **1**, No. 3, 26–49 (August/September 1998).
15. R. Prins, A. Blokdijk, and N. E. van Oosterom, "Family Traits in Business Objects and Their Applications," *IBM Systems Journal* **36**, No. 1, 12–31 (1997).
16. *IAA Product Builder: Language Reference Manual*, LB-14-0299-00, IBM Corporation (1997).

Accepted for publication December 30, 1999.

David Bevington *IBM United Kingdom Ltd., Hursley Park, Winchester, Hants S021 2JN, United Kingdom (electronic mail: david_bevington@uk.ibm.com).* Mr. Bevington received a B.Sc. degree in mathematics and physics from McGill University. He joined IBM's internal information technology organization and

held a number of technical and management positions. He is currently working as a business analyst in IBM Global Services in the United Kingdom, where he has been responsible for the business function of a number of major custom application developments. His interests are in making custom application development more of an engineering discipline, and in techniques for building applications with maximum inherent flexibility.