<table>
<thead>
<tr>
<th>(51) International Patent Classification</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>G06F 9/44</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(21) International Application Number:</th>
<th>PCT/CA97/00330</th>
</tr>
</thead>
<tbody>
<tr>
<td>(22) International Filing Date:</td>
<td>15 May 1997 (15.05.97)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(30) Priority Data:</th>
</tr>
</thead>
<tbody>
<tr>
<td>60/017,994 20 May 1996 (20.05.96) US</td>
</tr>
<tr>
<td>08/856,952 15 May 1997 (15.05.97) US</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(71) Applicant:</th>
</tr>
</thead>
<tbody>
<tr>
<td>COGNOS INCORPORATED (CA/CA); 3755 Riverside Drive, Ottawa, Ontario K1G 3Z4 (CA).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(72) Inventors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODEN, Colin, Andrew; 5 Steele Chase Drive, Kanata, Ontario K1G 3Z4 (CA). HIROSE, Stan; 350 Kanrack Avenue, Ottawa, Ontario K2A 0A2 (CA). GIBB, Robert; 533 Mansfield Avenue, Ottawa, Ontario K2A 2S8 (CA). MOFFAT, Eric; Apartment 1715, 1375 Prince of Wales Drive, Ottawa, Ontario K2C 3L5 (CA).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(74) Agents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>O’NEILL, T., Gary et al.; Gowling, Strathy &amp; Henderson, Suite 2600, 160 Elgin Street, Ottawa, Ontario K1P 1C3 (CA).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(54) Title:</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLICATION SERVICES BUILDER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(57) Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>An application program builder having a multitude of predetermined high-level objects, a plurality of predetermined templates for building corresponding programs based upon the high-level objects and a workbench interface for providing access to and manipulation of the templates and high-level objects is provided. The high-level objects are able to abstract the business developer from the technical implementation details, allowing the developer to work at a business level rather than an implementation level.</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>AL</td>
</tr>
<tr>
<td>AM</td>
</tr>
<tr>
<td>AT</td>
</tr>
<tr>
<td>AU</td>
</tr>
<tr>
<td>AZ</td>
</tr>
<tr>
<td>BA</td>
</tr>
<tr>
<td>BB</td>
</tr>
<tr>
<td>BE</td>
</tr>
<tr>
<td>BF</td>
</tr>
<tr>
<td>BG</td>
</tr>
<tr>
<td>BJ</td>
</tr>
<tr>
<td>BR</td>
</tr>
<tr>
<td>BY</td>
</tr>
<tr>
<td>CA</td>
</tr>
<tr>
<td>CF</td>
</tr>
<tr>
<td>CG</td>
</tr>
<tr>
<td>CH</td>
</tr>
<tr>
<td>CI</td>
</tr>
<tr>
<td>CM</td>
</tr>
<tr>
<td>CN</td>
</tr>
<tr>
<td>CU</td>
</tr>
<tr>
<td>CZ</td>
</tr>
<tr>
<td>DE</td>
</tr>
<tr>
<td>DK</td>
</tr>
<tr>
<td>EE</td>
</tr>
<tr>
<td>EL</td>
</tr>
<tr>
<td>EM</td>
</tr>
<tr>
<td>EO</td>
</tr>
<tr>
<td>EE</td>
</tr>
<tr>
<td>EN</td>
</tr>
<tr>
<td>ER</td>
</tr>
<tr>
<td>ES</td>
</tr>
<tr>
<td>ET</td>
</tr>
<tr>
<td>EU</td>
</tr>
<tr>
<td>FI</td>
</tr>
<tr>
<td>FJ</td>
</tr>
<tr>
<td>FK</td>
</tr>
<tr>
<td>FO</td>
</tr>
<tr>
<td>FR</td>
</tr>
<tr>
<td>GA</td>
</tr>
<tr>
<td>GD</td>
</tr>
<tr>
<td>GE</td>
</tr>
<tr>
<td>GH</td>
</tr>
<tr>
<td>GI</td>
</tr>
<tr>
<td>GL</td>
</tr>
<tr>
<td>GM</td>
</tr>
<tr>
<td>GN</td>
</tr>
<tr>
<td>GO</td>
</tr>
<tr>
<td>GR</td>
</tr>
<tr>
<td>HK</td>
</tr>
<tr>
<td>HM</td>
</tr>
<tr>
<td>HN</td>
</tr>
<tr>
<td>HR</td>
</tr>
<tr>
<td>HT</td>
</tr>
<tr>
<td>HI</td>
</tr>
<tr>
<td>HM</td>
</tr>
<tr>
<td>HG</td>
</tr>
<tr>
<td>HR</td>
</tr>
<tr>
<td>HI</td>
</tr>
<tr>
<td>HI</td>
</tr>
<tr>
<td>HR</td>
</tr>
<tr>
<td>HT</td>
</tr>
<tr>
<td>HG</td>
</tr>
<tr>
<td>HI</td>
</tr>
<tr>
<td>HR</td>
</tr>
</tbody>
</table>
APPLICATION SERVICES BUILDER

FIELD OF THE INVENTION

The present invention relates to an applications services builder and is particularly concerned with sophisticated business applications for utilizing information from databases.

BACKGROUND OF THE INVENTION

The desirability of accessing database information is well established. The development of such application programs has often been by traditional programming methods, that is via custom development of applications. Large entities, with sophisticated information management departments, have traditionally had the necessary expertise for this type of application development activity in client/server environments. However, it is desirable for smaller companies to have access to greater sophistication in developing their business applications, while recognizing limitations imposed by resource availability.

Competitive advantage has always been the driving force in information technology trends. The early information age was driven by pioneering organizations who saw that by applying automation technology to core business practices, they could gain an operational advantage over their competitors. Over the past several decades, all businesses have computerized their core practices, thus eliminating the competitive advantage. By the 1990s organization had begun to look for ways to make the information, generated by automation, available to a new class of employees known as "knowledge workers". Giving these workers more control over business information requires decentralization of computing facilities. This need led to the client/server paradigm. Some businesses took a further step leading to the "Business Intelligence" boom.

The next competitive push is for the ability of organizations to respond to their customers and market by moving beyond decision support to decision implementation. An early era of applications development responded to the
need to automate business procedures and was driven by time-shared
minicomputers and the availability of development languages that were
accessible to business. Further developments were driven by a desire for
information access and facilitated by the relation database and client/server
technologies. Easy-to-use GUI tools and specialized business intelligence
software, which made decision support accessible to a wide range of
organizations, contributed to this trend.

The industry has defined a logical application architecture that breaks
application processing down into three service levels: User Services, Business
Services, and Data Services. In the environments for competitive
applications, the tools for providing users services and data services are well
established. Because these services are de facto standard and pervasive, they
have become commodities. Though an essential part of the environment,
these services provide no competitive differentiation to business.

Similarly, Windows and Internet platforms define a physical client
server model that can be used to define a physical application architecture.
The technology for providing the physical connectivity between the layers is
well established through communication and networking standards. The
connectivity environment is also a commodity that provides no competitive
differentiation to business.

There are a number of technical solutions available for building
sophisticated applications, that capture your unique business knowledge or can
be developed quickly by a business-oriented developer using a component
architecture on both Windows and Internet platforms. But none of these
solutions meet all these requirements.

There are many pre-built business application packages that are highly
suitable solutions for core applications. But, by their very nature, they cannot
represent your unique business knowledge and therefore limit your ability to
differentiate and create competitive advantage.

GUI builders are excellent for building sophisticated user interfaces
accessing data from a simple data source, but competitive advantage does not
come from the application user interface or the physical data storage. It comes from the business procedures and information that lie between the user interface and the data.

Similarly, HTML pages, accessed through a WEB browser, are excellent for information publishing and for simple business processes, but competitive advantage goes beyond information distribution.

The high-end client server tools and 3GL languages such as C++ offer the promise of sophistication, but these languages are not accessible to the business developer and force an understanding of object-oriented paradigms that go beyond the results focus needed for competitive application. These tools do not address the business requirement of delivering rapidly and being responsive to business change.

Competitive applications must be delivered by business developers in a business climate that is both immediate and changing. This requires a focus on the business result, not the enabling technology. Tools that are too hard or complex or that focus on the wrong portion of the application work as barriers to the effective delivery of applications.

**SUMMARY OF THE INVENTION**

An object of the present invention is to provide an improved application services builder.

Application services represent that part of the logical application architecture where competitive advantage is created. They represent the business service functionality of a competitive application that lies between the user services and the data services and represents a logical part of the application.

Applications that provide competitive advantage are not simple applications. To allow decision implementation they require access to multiple information sources and the specification of comprehensive business procedures. They must access and update information across the network, potentially crossing corporate boundaries.
The requirement for these sophisticated application services has arisen as a direct result of organizations transforming from a functional-based structure to a business process-oriented model.

Competitive applications are business-driven and specialized for one's unique way of doing business. And one's unique way of doing business is embodied in the sophisticated business information and business processes that represent one's business.

By nature, the factors driving competitive advantage are immediate and changing. Notwithstanding the sophisticated nature of competitive applications, they must be delivered as quickly as possible to meet immediate demand and adapt painlessly as business demands change.

Because competitive applications embody unique business knowledge, they must be delivered by business-oriented developers who have both an intimate understanding of the business and the skills to exploit the enabling technology. Business developers, typically, have a results-first mentality and must not be constrained by difficult-to-use technologies that focus on the technical solution and not on the business result.

Application services create competitive advantage by turning data into information. By understanding the relationships between data sources from a business point of view, application services enable a customer and all their related information to be viewed and manipulated as a single entity, not as a collection of tables that need to be joined.

Application services create competitive advantage by turning human decisions into application actions and procedures. By understanding the business policies that define how various actions interrelate, a decision to create a new customer may result in a number of procedures, such as checking credit, updating the marketing database, as well as adding the necessary customer information.

Most importantly, application services represent one's business policies and rules and therefore define one's competitive advantage.

A tool becomes accessible to a business developer when it deals with
the developer in terms of business abstractions as opposed to technical implementations. It must leverage market skills rather than requiring retraining of business knowledgeable staff. It must provide productivity by allowing the developer to focus on the end result rather than on the fact that the business components in the application will need to be recycled as business needs evolve.

The present invention is specialized for building and managing application services. The present invention allows the business developer to build and maintain competitive advantage through application services.

Application services represent components in the physical application architecture. These components provide a complete set of related business processing that is independent of both the user services driving the application and the data sources storing the application data.

The present invention is designed for the business developer and includes objects that abstract the business developer from technical implementation details, allowing the developer to work at a business level rather than an implementation level. These objects are those required to capture business knowledge and to build complete application services. The objects combine with industry-standard user interfaces and programming models to exploit existing skill sets and increase the productivity of the developer. Inheritance combines with program templates and wizards to allow applications to be built quickly and effectively and to allow business knowledge and processes to be accumulated and reused as application needs change.

The present invention comprises high-level objects, including a data access model, linkages, data transactions, record sets and fields, specialized for implementing application services. At the highest level these objects can be specialized through properties and events to capture one’s unique business information, such as business data relationships, the series of actions associated with processing a user request, or one’s unique business rules. Within an application services module, these objects cooperate to implement
the details of the business process described by the properties and events. And at runtime, they provide the technical implementation to coordiantae with multiple heterogenous data service components.

Because the application services objects are implemented separately from the presentation objects, business knowledge can be captured and implemented separately from the user interface.

The present invention includes other business objects, such as elements, tables, table relationships, and style forms that capture business knowledge. This information is reused as applications are built.

The present invention includes component objects such as templates, programs, modules, forms, menus, and status lines that facilitate building complete application service components.

The present invention addresses the needs of the business developer, allowing applications to be developed quickly and efficiently by teams of business knowledgeable developers.

The present invention uses visual programming techniques including an object explorer and a standard object programming language a employing familiar programming model, e.g. object, method, property, event). It is not another new programming language to learn, but a set of objects that insulate the business developers from implementation details by representing business functionality.

**BRIEF DESCRIPTION OF DRAWINGS**

The present invention will be further understood from the detailed description, together with the following drawings in which:

Fig. 1 illustrates, in a functional block diagram, a high-level view of the application program builder in accordance with an embodiment of the present invention;

Fig. 2 illustrates, in a high-level block diagram, a system based on an application built using the embodiments of the present invention of Fig. 1;

Fig. 3 illustrates a library object of Fig. 2;
Fig. 4 illustrates a relational database object of Fig. 2;
Fig. 5 illustrates an application object of Fig. 2;
Fig. 6 illustrates a program object of Fig. 5;
Fig. 7 illustrates a presentation module of Fig. 6;
Fig. 8 illustrates an application module of Fig. 6;
Fig. 9 illustrates a data connection module of Fig. 6;
Fig. 10 illustrates a workspace and a repository in accordance with the
embodiment of Fig. 1;
Fig. 11 illustrates an explorer window for the workbench of Fig. 1;
Fig. 12 illustrates, in a schematic diagram, an editing via popup
technique for the template object of Fig. 3;
Fig. 13 illustrates, in a schematic diagram, an adding via popup
technique for the template object of Fig. 3
Fig. 14 illustrates, in a schematic diagram, an add if key changed
technique for the template object of Fig. 3;
Fig. 15 illustrates, in a schematic diagram, an add if not found
technique for the template object of Fig. 3
Fig. 16 illustrates, in a schematic diagram, a grid technique for the
template object of Fig. 3;
Fig. 17 illustrates, in a schematic diagram, a VCR controls technique
for the template object of Fig. 3
Figs. 18a and b illustrate, in a schematic diagram, parallel details
technique for the template object of Fig. 3;
Fig. 19 illustrates, in a schematic diagram, tab pages embodiment of
parallel details technique of Figs. 18a and 18b;
Fig. 20 illustrates, in a schematic diagram, multiple windows
embodiment of parallel details technique of Figs. 18a and 18b;
Fig. 21 illustrates, in a schematic diagram, form regions embodiment
of parallel details technique of Figs. 18a and 18b;
Fig. 22 illustrates, in a schematic diagram, independent embodiment of
parallel details technique of Figs. 18a and 18b;
Fig. 23 illustrates, in a schematic diagram, an insert a template dialog box in accordance with an embodiment of the present invention;

Fig. 24 illustrates, in a schematic diagram, a template wizard in accordance with an embodiment of the present invention;

Fig. 25 illustrates, in a schematic diagram, a pick a template panel in accordance with an embodiment of the present invention;

Fig. 26 illustrates, in a schematic diagram, a data model panel in accordance with an embodiment of the present invention;

Fig. 27 illustrates, in a schematic diagram, a techniques panel in accordance with an embodiment of the present invention;

Figs. 28 illustrates, in a schematic diagram, a form panel in accordance with an embodiment of the present invention;

Fig. 29 illustrates, in a schematic diagram, a what next dialog box in accordance with an embodiment of the present invention;

Fig. 30 illustrates, in a Booch diagram, a normal inheritance in accordance an embodiment of the present invention;

Fig. 31 illustrates, in a Booch diagram, list inheritance in accordance an embodiment of the present invention;

Fig. 32 illustrates, in a Booch diagram, changing inherited object list of Fig. 31;

Fig. 33 illustrates, in a Booch diagram, changing the inherited parent of Fig. 31;

Fig. 34 illustrates, in a Booch diagram, extending the inherited list of Fig. 31;

Fig. 35 illustrates, in a Booch diagram, normal inheritance with overriding of property value in accordance an embodiment of the present invention;

Fig. 36 illustrates, in a flow chart, a method of application program building in accordance with an embodiment of the present invention;

Fig. 37 illustrates, in a block diagram, the user interface framework for the workbench of Fig. 1;
Fig. 38 illustrates, in a block diagram, the framework class hierarchy for the user interface framework of Fig. 37;

Fig. 39 illustrates, in a block diagram, an instantiation tree for a running UI framework application;

Fig. 40 illustrates, in a block diagram, look up operations on the common object cache;

Fig. 41 illustrates, in a block diagram, a registration process in the user interface framework; and

Fig. 42 illustrates, in a block diagram, an object look-up in the common object cache.

DETAILED DESCRIPTION

Referring to Fig. 1, there is illustrated in a functional block diagram, a high-level view of the application services builder in accordance with a embodiment of the present invention. The application services builder 10 includes a workbench user interface 12, templates 14, high level objects 16 and instance inheritance 18.

In building an application services, a user through the workbench user interface 12 creates, duplicates, and modifies objects 16 either directly or indirectly through the use of templates 14 and through the use of instance inheritance 18, existing objects can be reused.

The high-level objects 16 include a data access model, linkages, data transactions, record sets and fields, specialized for implementing application services. At the highest level these objects can be specialized through properties and events to capture one’s unique business information, such as business data relationships, the series of actions associated with processing a user request, or one’s unique business rules. Within an application services module, these objects cooperate to implement the details of the business process described by the properties and events. And at runtime, they provide the technical implementation to coordinate with multiple heterogenous data service components.
Templates 14 reflect patterns of standard business processing behaviour, and Wizards apply this behaviour to multiple data sources, automatically creating components made up of many cooperating objects. The resulting programs already contain the business rules and data relationships defined in the present invention.

While other tools take an "assemble-from-parts" approach to program creation, the present invention uses a "results first" approach.

Referring to Fig. 2, there is illustrated a system based on an application service developed using the embodiment of Fig. 1. The system 20 includes an application 22, a database 24, and a library 26. High-level links are shown with arrows. The database 24 is linked to the library 26. The application 22 is linked to both the database 22 and the library 26. The links allow application database and library objects to share information. While each of the high-level objects contain other objects, linking is only necessary between high-level objects. Linking is used to define relationships between high-level objects, and is applied through the workbench user interface 12. Linking indicates that information can pass from one object to another object, and , applied to high-level objects, defines the organization of system 20.

Through a shared repository of the present invention, business developers can leverage the work of others, as they build competitive applications together.

The present invention addresses the needs of applications that must adapt as business demands change. And offers a number of ways that business knowledge can be reused and recycled as the business evolves.

Libraries of the present invention organize business knowledge as it is captured. Through inheritance this business knowledge can be used and reused in multiple programs implementing application services. As additional application services modules are developed, they too can be put into libraries and their business content used for subsequent application services.

The present invention allows application services to be assembled into programs depending on the personal needs of business users. Then
application services components can be packaged and assembled into components as the needs of the business dictate.

Component architecture also lays the foundation for application services that can be used to rapidly respond to business change. Over time, component reuse and recycling will lower development costs and facilitate business integrity and consistency.

The Windows platform provides a number of opportunities for implementing components including executable, shared libraries (DLLs), OLE Custom Controls (OCX) and OLE Automation Servers. Application services, through their independence, can take advantage of the various technologies and distribution and sharing capabilities made available to different classes of components. Through these various component technologies, application services can be designed to run on the client or on the server.

The Internet/Web platform also offers a number of opportunities for implementing application services based on components. This platform and its HTML page architecture is being rapidly adopted for information distribution. As the environment continues to evolve, it is expected that distributable components in the form of agents will facilitate action-oriented services.

A competitive application can be made up of a number of components. The application service component represents the business processing implementing the competitive advantage. Other components can implement data services, user services or other business or application services of the application.

Application services have an architecture that provides integration with other components. This integration occurs at the business level rather than at the technical level. For example, when an application service uses SQL and ODBC to talk to a data service, and the SQL DML call made are handled by objects implementing the application service.

Now that an overview has been given of the structure and operation of the application program builder has been given, a more detailed description of the components thereof is provided.
Referring to Fig. 3, there is illustrated the library object. The library object 50 is a container object. While not part of the application program, the objects contained in the library object are used to build the application program and for instance inheritance, both simplifying the application program building process. The library object 50 includes several objects that facilitate business application program building. These objects include an element object 52, a structure object 54, a form object 56, a style form object 58, a template object 60 and module objects for application 66, presentation 68, and data connection 70. A typical library contains many objects of each type.

Referring to Fig. 4, there is illustrated a database object. The database object 80 is a container object and includes two types of objects, table objects 82 and view objects 84. When data definitions are imported, many such objects are produced to store the imported data in object form.

Referring to Fig. 5, there is illustrated an application object 90. The application object 90 includes several program objects, for example programs 92, 94 and 96. The number of program objects depends upon the complexity of the application program.

Referring to Fig. 6, there is illustrated an application program object 100. The application program object 100 includes a presentation module 102, an application module 104, and a data connection module 106. The type of a module is determined by objects contained therein. Typically, more than one of each type of module is contained in an application program object. The modules are representative of business processing modules.

Referring to Fig. 7, there is illustrated the presentation module of Fig. 6. The presentation module contains objects associated with building an interactive interface to a program. The presentation module includes procedures and form 110, toolbar 112, menu 114, and status line 116 and action 118 objects. The form object 110 includes control and data control objects 120 and 122.

The form object 110 is the object used to create windows in a program. Form is contained within a presentation module and as such can
only be manipulated within the context of a presentation module.

Referring to Fig. 8, there is illustrated the application module of Fig. 6. The application module 104 contains objects associated with managing an application database with an application program. In addition to procedures, the application module 104 includes data access model 130, data transaction 132, record set 134, and SQL request 135 objects. The data access model includes linkage objects 136. The record set object 134 includes field objects 138. The data access model 130 is the only one in an application module 104. The purpose of the data access model 130 is to define the relationship between the root source record set and all its target record sets. It also provides global properties relative to the record set. The linkage object 136 is a role object that defines how the source record set participates in a relationship with the target record set. Linkages and record sets are the basic building blocks of the data access model. The record set 134 provides the data and the linkage 136 defines how the data is related to and how it is processed relative to actions occurring in the source record set. The record set 134 is a set of qualified rows based on some criteria. The data in a set is accessible to the user as if it were loaded in memory regardless of implementation. A user moves through the record set by moving the current row indicator. The field 138 represents a single data item in the current row of a record set 134. Fields 138 do not exist outside the record set 134 and may only be referenced through a record set reference. The SQL request object 135 handles singleton requests that do not produce a result set. The SQL request object also supports the execution of parameterized requests.

The data transaction 132 represents logical transactions not physical ones. By default there is only one data transaction 132 per application module 104.

Referring to Fig. 9 there is illustrated the data connection module of Fig. 6. The data connection module 106 includes a data connection object 140. The data connection object 140 represents a run-time associated to a liquid database. This database may be a relational database, a directory of
files, or a set of two or more relational databases.

Application Data: One of the main purposes of a program is to allow
the user to view and manipulate application data. All application data in a
program is represented as a field object within a RecordSet object.

RecordSet: The standard RecordSet object is used to interact with
application data in application databases.

Field: The field object is used to manipulate a single data item within
a record set. The field object represents the value and datatype of the
application data and also has events and methods that are used during
manipulation of the data.

Each module can contain a single ModuleRecord which is used to
manipulate transient data within the program.

Binding Controls To Application Data: Application data is bound to
the controls on a form through the DataControl object. The DataSource
property of a bound control specifies a DataControl and the DataField
property defines a field in the RecordSet associated with DataControl. When
a control is bound to application data it determines its value based on the
current value of the current row of the associated record set. Operations
against the RecordSet object the filed object and the DataControl object can
affect this value.

DataControl Processing: The DataControl object is used to relate a
row in a record set to the controls and control extensions associated with
fields in the RecordSet. It is a row level interface to the record set. The data
control itself has no data. It coordinates the interchange of information
between the RecordSet and associated controls and control extensions. The
data control can also have a "VCR" style interface which the user can use to
change the current row of the RecordSet.

The Connect property and the RecordSource property are used, at
development time, to specify the RecordSet which supplies data to the data
control. these two properties are used at runtime to determine the value of the
runtime RecordSet property.
The RecordSet property of the DataControl is used to reference a specific RecordSet instance which supplies the data to this data control. This property is a runtime property which is determined from the Connect and RecordSource properties of the data control. It can be overwritten at runtime to point to a specific instance of a record set. The IndependentRecordControl and CurrentRow properties of the DataControl can be used to allow the DataControl to maintain the current record to be displayed, rather than relying on the underlying RecordSet. The default value of IndependentRecordControl is False. In this case, the CurrentRow property is determined from the recordSet. When this property is set to True, the DataControl maintains its own CurrentRow counter.

Null Valued Data: Data from fields, which have a Null value, are displayed using the NullCharacter value for the appropriate datatype. The DateNullCharacter, StringNullCharacter and NumericNullCharacter of the program object define the character to use. These values are determined at development time and are not changeable at runtime.

Deleted Data: Individual controls for fields which are displaying data from a deleted record in a record set are disabled. Cells in a grid Control, which are displaying data from a deleted record in a record set, are displayed in a grayed fashion. These cells will not be disabled, but input to these fields will be ignored.

Multiple Views Of Data: Independent forms reflecting the same data. By default, multiple forms referencing the same record set instance will be synchronized so that the current record of the record set will be used to indicate the current record used by the data control.

Multiple Instances Of Form Representing Record Set Instancee: Forms are implemented using the ContextSet object. When a Form Instance is created the RecordSet property of the datacontrol must be set to the RecordSet instance that is to be displayed.

Multiple Instances Of Form Representing Different Records: Forms are implemented using the ContextSet object. The IndependentRecordControl
property of the DataControl object is used to indicate that the DataControl will maintain its own CurrentRow number. The CurrentRecord can be set when the Form is created.

Referring to Fig. 11, there is illustrated in a block diagram a development environment including a workspace 150 and a repository 152 in accordance with an embodiment of the present invention. In preparing to create an application, a user first creates a workspace 150 and links it to a repository 152.

Referring to Fig. 12, there is schematically illustrated an explorer window 154 for the workbench interface of Fig. 1. The explorer window 154 includes a left panel 156 and a right panel 158. In creating an application, a user accesses objects via the explorer window 154.

Templates are a mechanism for achieving high speed program creation. The three template classes are defined by a particular subsets of technique choices, chosen from the following list:

- Edit via Popup
- Add via Popup
- Add if Key Changed
- Add if Not Found
- Grid
- VCR Controls
- Parallel Details
- Multiple Detail Windows
- Search
- Status Panels
- Tool Bars
- Menus
- Inherits From
- Object Styles Form

Referring to Fig. 12 there is illustrated, in a schematic diagram, an editing via popup technique for the template object of Fig. 3:
Editing via Popup  Values: TRUE/FALSE

This technique choice determines if data changes are applied in place on a main form 160, or in a special popup edit window 162. When this is set to True a second form specifically for editing must also be generated. This is visually a copy of the main form 160 except that it is modal. The popup edit form 162 has OK 164 and Cancel 166 buttons generated on it in the top right corner. The OK button 164 closes the popup window, transfers all changes made in controls in the popup window 162 to the equivalent controls in the main form 160, and invokes the "update" Action. The Cancel button 166 closes the popup window, abandoning all changes made in the popup window. The edit controls on the popup form 162 are editable, the edit controls on the main form are read-only.

An action object called "Edit" is generated that shows the popup window and populates its data controls with copies of the values currently in the record set. This assumes that the module and form templates being used are prepopulated with button, tool or menu entry that calls the "Edit" Action. The Module and Form templates include these.

Referring to Fig. 13 there is illustrated, in a schematic diagram, an adding via popup technique for the template object of Fig. 3.

Adding via Popup  Values: TRUE/FALSE

Very similar to the above edit technique question, however the developer may have a requirement to support adding new records as a separate style then editing existing information. When this is set to True a second form 170 specifically for adding is also generated. This is identical to the form used for Popup Editing, including the use of OK and Cancel buttons. In this case the "Add" Action object is generated that shows the popup window and populates its data controls with default or null values the record set.

It is optional whether the Add Action invokes the recordSet AddRow method, or whether this is delayed until the OK button is pressed. The combination of Add and OK creates a new record and makes that the current row. The combination of Add and Cancel is a null operation, i.e. the final
result is no extra row created and focus remains on the previous row. This assumes that the module and form templates being used are prepopulated with a button, tool or menu entry which calls the "Add" Action. The Module and Form templates include these.

Referring to Fig. 14 there is illustrated, in a schematic diagram, an add if key changed technique for the template object of Fig. 3. Add if Key Changed Values: TRUE/FALSE. If this flag is TRUE a new record 172 is automatically created from the current record 174, if the user changes the key. Changing the value of the Key field is taken to imply that a new record is added rather than changing the value of the key in the current record. This follows from the assumptions that key field values uniquely identify each row and that the key fields in a row should never be changed.

The implementation requires generation of a AddRecordOnKeyChange user defined procedure. This procedure is invoked when the user changes the value of an edit control, which is bound to a column that is part of the primary Key for the RecordSet's Main Table.

This procedure needs to:

1. Make sure that the new value in the edit control does not propagate to the field on the current Row.

2. Add a new row to the recordSet.

3. Copy the new value in the changed edit control to the equivalent Field in the newly added row.

4. Copy all other column values from the previous current row to the newly added row.

5. Change the current Row to be the newly added row.

Referring to Fig. 15 there is illustrated, in a schematic diagram, an add if not found technique for the template object of Fig. 3. Add If Not Found Values: TRUE/FALSE. If the user attempts to find a record and fails (i.e., it's not there) automatically go into adding a new record via which ever adding technique is active. For True: Some of the Search techniques allow users to attempt to find data by typing in a value for some key or indexed
field, like Employee Name. If this technique is set to True, and no Employee is found whose name (or whatever fields were specified) exactly matches the values given, then a new record 176 is added to the Record set, and eventually to the database. The new record should be prepopulated with the values which were specified for the retrieval.

From a logic perspective this requires the Find action to test whether the recordCount is O after the record set is executed, and if it is invoke the Add action. The control values specified by the user would also have to be cached somewhere so that they can be copies into the newly added row.

Referring to Fig. 16 there is illustrated, in a schematic diagram, a grid technique for the template object of Fig. 3. Grid Values: TRUE/FALSE. This is the choice of single vs. repeating primary on a form. True equates to the use of Grids. Instead of a set of edit controls the data is presented as a grid 178, with one grid row per record set row. Clearly this allows the user to see several data rows at the same time. The vertical height of the grid is as large as the data area on the form will allow.

With the interface, a vertical scroll bar 180 on the grid is the main data navigation technique since it allows the user to move back and forth between the data rows.

Referring to Fig. 17 there is illustrated, in a schematic diagram, a VCR controls technique for the template object of Fig. 3. VCR Controls Values: TRUE/FALSE. This defines whether the data control for updatable tables are shown as a visible VCR style controls 182 or hidden. This cannot be inherited since the number of them is data dependent.

For True: The Data Control for the table is included as a visible object on the form. It is positioned underneath the generated data controls. The VCR style controls 182 provided by the data control can then be the main mechanism for navigating between rows in the record set.

Referring to Figs. 18a and b there are illustrated, in a block diagrams, parallel details technique for the template object of Fig. 3.

Parallel details: values: Tab Pages/ Multiple Windows/ Form Regions
Parallel details occurs when one record set is the source of two linkages, both with a cardinality of many. Fig. 18a shows that customers 182 can have many orders 184 and many sites 186. But there is no relationship between Sites 186 and Orders 184. Therefore if one is viewing a Customers record, one UI is needed to view its sites and a separate UI is needed to look at its Orders, with neither taking precedence over the other.

Fig. 18b shows this situation extended to a more complicated example. In Fig. 18b, there are two parallel detail situations. The first is a 3-way parallel between sites 186, orders 184 and visits 188. The second is order lines 190 and deliveries 192 as parallel details of orders. (Organizations 194 is linked with cardinality one, hence it can be considered a horizontal extention of customers.)

Referring to Fig. 19 there is illustrated, in a schematic diagram, a tab pages embodiment of parallel details technique of Figs. 18a and 18b.

Referring to Fig. 20 there is illustrated, in a schematic diagram, multiple windows embodiment of parallel details technique of Figs. 18a and 18b.

Referring to Fig. 21 there is illustrated, in a schematic diagram, form regions embodiment of parallel details technique of Figs. 18a and 18b.

Referring to Fig. 22 there is illustrated, in a schematic diagram, independent detail window embodiment. Independent Detail Windows Values: TRUE/FALSE

This choice is about whether the application wants to treat details as independent objects. For True: If the value is True then the preprogramming causes the data in the detail form to be locked to the cursor on the selector form. Drilling into a detail the first time causes a detail window to be created. Subsequently moving the cursor on the selector form causes the data in the detail form to change to reflect the record pointed to by the cursor. This, is achieved by the use of a dependent linkage, and dependent data control on the details form.

In this model it makes little sense to have multiple detail windows,
since they would be focused on the same record. So the Detail Forms and Record sets do not need to be in a separate Context set.

**Search Values:** Popup/Modal form/Drop List/PreRetrieve

This technique defines the interface style used for searching and retrieving data rows for the root table. The meanings of the different choices are:

**Popup**
- A popup dialogue allows custom search criteria.
- Modal form is what Quick does today.
- Automatic is the Quicken partial entry process.
- Drop list provides all possible choices in a control.
- PreRetrieve simply retrieves all relevant data as soon as the form is opened. The user then navigates backwards & forwards through the record set using buttons, VCR controls or whatever.

**Inherits From Presentation Module, Values:** Presentation Module.

This technique is really just a reference to a presentation module, where a set of status lines, toolbars and menus are defined. The presentation module and as a result include copies of its Status Lines, Menus and Toolbars.

**Inherits From Application Module Values:** Application Module.

This technique is really just a reference to an application module where a set of user defined functions and other objects may be defined. The application module objects created by the template will inherit from this application module and as a result include copies of its objects.

**Inherits From Data Connection Module Values:** Data Connection Module.

This technique is really just a reference to a data connection module where a set of user defined functions and other objects may be defined. The data connection module objects created by the template will inherit from this data connection module and as a result include copies of its objects.

**Inherits From Form Values:** Form.

This technique is really just a reference to a form where a set of objects are defined. All forms created by the template will have copies of all the objects in the inherited form. These
objects will inherit their properties from the objects contained in the inherited form. The form inherited is expected to define a standard background including things like Find & Update buttons, it is also expected to inherit form Style definition form.

There are 3 template classes. The development environment is prepopulated with at least one template based upon each template class. The three template classes are one update table per form, master detail, and reference list.

One Update Table per Form is the most flexible of the templates. It can take an arbitrary hierarchical data model, and derive a set of modules and forms that can browse, change, delete and add data. The preprogramming process uses the data model to determine the number of levels and the number of parallel detail branches that need to exist.

The information describing the selector will be displayed on the details form, but is not modifiable, it is there just to set the context.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit via Popup</td>
<td>Yes</td>
</tr>
<tr>
<td>Add via Popup</td>
<td>Yes</td>
</tr>
<tr>
<td>Add if Key Changed</td>
<td>Yes</td>
</tr>
<tr>
<td>Add if Not Found</td>
<td>Yes</td>
</tr>
<tr>
<td>Grid</td>
<td>Yes</td>
</tr>
<tr>
<td>VCR Controls</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallel Details</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple Detail Windows</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Master and Detail template has a main table and resulting details are shown on the same form. There is no provision for supporting more than two levels of information.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit via Popup</td>
<td>Yes</td>
</tr>
<tr>
<td>Add via Popup</td>
<td>Yes</td>
</tr>
<tr>
<td>Add if Key Changes</td>
<td>Yes</td>
</tr>
<tr>
<td>Add if Not Found</td>
<td>Yes</td>
</tr>
<tr>
<td>Grid</td>
<td>Yes</td>
</tr>
<tr>
<td>VCR Controls</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallel Details</td>
<td>Tabs/Regions only</td>
</tr>
<tr>
<td>Multiple Detail Windows</td>
<td>No</td>
</tr>
<tr>
<td>Search</td>
<td>Yes</td>
</tr>
<tr>
<td>Inherits from Module</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Reference List is a template type that allows a user to automatically create drop lists for maintaining one-to-one linkages. A typical example of this would be maintaining the branch information on an employee form using a drop list of the Branch Names rather than by entering and validating the branch code.

Most of the work is done at the record set level rather than at the form level. To maintain the employees to branches linkage this way would require:

1. A Branches record set which contains all the Branches records. This needs to be retrieved when the form starts up.

2. Logic which maintains the current row in the Branches record set at the row which matches the branch Code of the current Employees row. This work two ways. When the current Employees row moves, the current Branches row will need to be reset. Conversely, if the current Branches row is moved then the Branch Code on the current Employees row needs to be updated, this is how the relationship is maintained.

3. A drop list (or similar UI facility) which shows the current Branches row, typically by showing the Branch Name, and allows the current Branches row to be changed, typically by showing all the Branch names in the drop list and allowing them to be selected.

Referring to Fig. 23 there is illustrated, in a schematic diagram, an
insert a template dialog box used for template creation in accordance with an embodiment of the present invention. Templates are Explorer objects that are contained in Libraries. A new Template can be created by selecting "Template..." from the Insert menu, or from the template toolbar button.

In either case, the developer will be asked which class of template to create via a simple dialog box.

Template objects are maintained like any other Explorer Object. A user can rename and reorganize them within the Explorer, more detailed maintenance is achieved by launching a child window for the object from the Explorer.

The Template object window has two tabs. The first tab overlay is the identify tab page. This is a properties style overlay with the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Character String</th>
<th>The name that appears in the pick a template dialog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Class</td>
<td>&quot;Template&quot;</td>
<td>Read-only, as part of object identification</td>
</tr>
<tr>
<td>Class</td>
<td>Enumeration</td>
<td>The template class, a read only attribute (see Template Classes)</td>
</tr>
<tr>
<td>Reference</td>
<td>Single line text</td>
<td>Object reference ID for requirements traceability</td>
</tr>
<tr>
<td>Notes</td>
<td>Multi-line Character String</td>
<td>Developer Documentation area</td>
</tr>
<tr>
<td>Description</td>
<td>Multi-line Character String</td>
<td>The description displayed in the pick a template dialog</td>
</tr>
<tr>
<td>Picture</td>
<td>File Reference</td>
<td>The bitmap displayed in the pick a template dialog</td>
</tr>
</tbody>
</table>

The second tab page is the techniques tab page. This is a properties style overlay showing those Template techniques that are applicable to this class. The purpose of this overlay is to allow users to define the default values which will appear for these techniques when the Template Wizard is invoked.
Referring to Fig. 24 there is illustrated, in a schematic diagram, a template wizard in accordance with an embodiment of the present invention. The Template wizard can be invoked from the Insert menu options:
Interactive Event Program...: To create a new program and populate it.
Template Wizard...: To add information to an existing Program.

There is one Wizard tool, which adds information to an Interactive Event Program if one (or a child of one) is currently selected, and creates a new Interactive Event Program if the selection is not inside an Interactive Event Program.

The Wizard can also be invoked by dropping a number of tables and Relationships (selected on a table Relationships diagram) onto an Application or an Interactive event Program. If the drop is onto an Application, the Wizard is invoked to create a new Interactive Event Program. If the drop is onto an Interactive Event Program, the Wizard is invoked to add information to the Interactive Event Program. In either case, the Data Model panel will eventually appear prepopulated with the selected tables (as record sets) and Relationships (as Linkages). A requirement of the Data Model interface is that there is always a root Record Set. A workable heuristic would be to pick any table within the selection which has no many-to-one relationships to other tables. This will be the top of the selected data subtree.

To Match the Microsoft guidelines, the caption will be either "Insert Interactive Event Program" or "Template Wizard" to match the context the dialog is invoked from.

Back Button 200 moves to the previous panel in the sequence. When the user is on the first panel in the sequence, this button is disabled.

Next Button 202 moves to the next panel in the sequence. When the user is on the last panel in the sequence, this button changes its label to "Finish". This can be context dependent. If the user is in a state where he cannot meaningfully move to the next panel without entering more data, this button should be enabled, but when pressed popup a message which explains why Next is not currently valid.
Cancel Button 204 exists from the Wizard without creating any objects.

Help Button 206 invokes a help topic for the current Wizard panel.

Referring to Fig. 25 there is illustrated, in a schematic diagram, a pick a template panel in accordance with an embodiment of the present invention.

Module Wizard. This is the first panel that appears. The pick a template panel includes a template list and edit box 208, a picture area 210, a description area 212 in addition to back, next, cancel, and help buttons.

The template list and edit box 208 lists the templates in the selected library and allows one to be selected. The edit box at the top of the list is used in a similar way to the Styles list in the Style gallery dialog in Microsoft Word. Firstly, the user can enter a series of letters here that cause the list to scroll to put the first template that starts with those letters at the top of the list. Secondly, when the user selects a template on the list, its name is echoes in the edit box.

The list is sorted alphabetically, which will make the additional choice "<none>" sort after most meaningful names. Each Library has a property that is the default template. This template will be selected automatically when the library is chosen, implicitly this includes when the dialog is first opened. If this property is not set, the first template on the list is selected by default.

When this dialog is called as part of a Interactive Event Program insert there should be a <none> option in the list of templates, so that the developer can create an empty Interactive Event Program.

Double-clicking on a list entry will select the template and "press" the default button, which is Next.

The picture area 210 shows a bitmap that is a property of the selected Template. This bitmap is a diagram or schematic that illustrates the kind of program/modules the template will create. User testing so far has shown that thumbnails of the Forms produced are more recognized that data access diagrams.

The description area 212 shows a long text string that is a property of the selected Template. This text explains the kind of program/modules that
the template will create.

Referring to Fig. 26 there is illustrated, in a schematic diagram, a data model panel in accordance with an embodiment of the present invention. The data model panel includes a data model diagram 214 and insert table 216, set root 218 and remove 220 buttons in addition to the back, next, cancel and help buttons. The data model panel is used to define the data module object in the resulting data module. Templates of the navigation Class will skip this panel since they are not data driven.

The data model diagram 214 is the method of defining the data model. The main mechanism for adding to this diagram is to drag tables and relationships from table relationship diagrams, which is why this dialog is non-modal.

The data model must have a root. The record set that is the root is the update table of the first form record, and much of the information used to drive the template creation process is built from knowing which record is the root record.

The root record set can be specified by double-clicking on a record set node, if there was previously a root record set defined the root designation is switched to the node which was double-clicked. For the Master and Detail template Class, this diagram is limited to 2 levels. The algorithm for determining the default root record set needs to be consistent between this dialog and the Data Access Overlay of the Application Module window.

The insert table button 216 is enabled at all times. The insert table button 216 invokes a dialog which allows one or more tables to be picked from the databases connected to the target application. The selected tables should be added to the data model as record sets in the same way as if they were dropped onto the diagram. This dialog should have the same appearance and functionality as the other "Select a <thing>" dialogs (also known as Get-Ref dialogs throughout Axiant. In effect, this is the same dialog which needs to be invoked from the Insert Table menu option when in the data Access overlay of the Application Module Window.
The set root button 218 is enabled when exactly 1 record set is selected in the data access program. The set root button 218 sets the selected record set to be the designated root record set. This is an additional UI that achieves the same as double-clicking on the record set, but provides a greater visual cue.

The remove button 220 is enabled when 1 or more record sets and/or linkages are selected in the data access diagram. This button deletes the selected objects in the data access diagram. This is an additional UI to the "Del" key, but provides a greater visual cue. This button is designed to be consistent with other dialogs, such as the Application Deployment dialog that allows objects to be removed via a button as well as the "Del" key.

Referring to Fig. 27 there is illustrated, in a schematic diagram, a techniques panel in accordance with an embodiment of the present invention. The techniques panel appears before the Form Overlays because some of the answers will determine how many Forms will be created and hence how many Form overlays will appear. The techniques panel include plural technique controls 222, in addition to back, next, cancel, and help buttons.

The technique controls 222 allows technique names to be entered. The technique names are those techniques that are applicable to the selected template Class. The technique values will default to those specified in the selected Template object. The interface to each technique is consistent with the application services bulider properties lists. For enumerations (including True/False) this is a drop list, for object references (like UI Modules) this is a popup "select a <thing> dialog".

Referring to Figs. 28 there is illustrated, in a schematic diagram, a form panel in accordance with an embodiment of the present invention. The form panel includes heading 224, table 226, and columns 228 controls, >> button 230, form controls list 232, and grid 234 and multiple detail windows 236 check boxes.

The heading 224 indicates how many forms are being generated, and what the intended focus of the current form is. A typical heading will be
"Form 2 of 3: Focus on Table Employees".

When the Parallel Details technique is set to Tabs, one panel will appear for each tab overlay. The format of the title will then change to "Tab Overlay <j> on Form <n> of <m>: Focus on <table name> Table>", which may need to spread over 2 lines.

The table control 226 is a drop list, that shows only some of the record sets defined on the data model panel. Typically, this will show the focus table, plus any tables that are connected 1-1 from it, or 1-many to it, transitively. (In a normal diagram layout, this means all those to the right or above the focus table, but not the ones lower down that involve a one-to-many linkage).

The columns control 228 lists the columns of the table for the selected record set. This list supports multiple selection.

The >> button 230 copies the columns selected in the columns list across to the Form Controls list 232. This button always adds the columns to the bottom of the list, but there is also a drag and drop procedure that allows them to be inserted anywhere in the Form Controls List 232.

The Form Controls List 232 shows the columns for which edit controls (of some type) or grid columns will be created in the form. The template will pre-select the columns to be in this list, so users need only fine tune the selection.

This list supports drag and drop for reordering, since order is important in the generate forms.

A Preselection Algorithm interface allows the user to add columns to the field list and to remove columns from the field list. Based on observation of 'typical' forms, the heuristics are:

1. For the main, updatable table on the form, include all Columns.
2. For the Detail tables in a Master Detail form include all columns, except those which participate in the relationship to the master table (since these will have the same values for all visible rows).
3. Other tables will be linked one-to-one from the main or detail tables and can be regarded as reference tables. For these tables, include only the column flagged as the Reference Key (this feature is described further in the last section of this paper).

4. Order the fields such that:
   - The detail fields come after the master fields.
   - The reference Key fields come directly after the Master/Detail columns which participate in the relationship to the reference table.

The grid and multiple detail windows check boxes 234 and 236 default to the values specified in the techniques panel.

Referring to Fig. 29 there is illustrated, in a schematic diagram, a what next dialog box in accordance with an embodiment of the present invention. The What Next Dialog box is not considered part of the Wizard, but instead is a separate dialog which appears after the Wizard is complete. The distinction is that when this dialog appears, the Wizard generation has happened and it is not possible to use a Cancel button to avoid creating the objects. The what next dialog box includes an created objects list 240, and edit 242, build 244, run 246 and explore 248 buttons.

The objects created list 240 lists the new Explorer objects created. In this case, there will be two or more modules, and potentially a program. The modules created automatically appear in this list sorted by name within type and the first is automatically selected. If a program object is generated, it should be the first, and selected object.

The Edit button 242 is enabled when something in the objects created list 240 is selected. It closes this window and opens a child window for the selected object. The Edit button 242 is the default button.

The Build button 244 is enabled when something in the objects created list 240 is selected. It closes this window and builds the selected object. On Completion, the Explorer window 154 of Fig. 11 is active with the newly
created and built object selected.

The Run button 246 is enabled when something in the objects created list 240 is selected. It closes this window then builds and runs the selected object. On completion, the run time window for the newly built object will be active.

The Explore button 248 is always active. It closes this window and activates the Explorer Window. If an object is selected in the created objects list 240, then that object will be selected in the Explorer Window.

Generation Rules

Objects to be Created. Each template creates one UI Module. The number of forms created inside this UI Module is determined by the template class and the data model specified. The Popup Search template class may be an exception and may add information to a UI Module instead.

All but Navigate class templates will also create a data module. A data source module will be created when needed, i.e. when there is no existing data source module for the databases needed already in the program.

Object Creation Mechanisms. The process of going from a template to the resulting modules and forms requires a number of steps and several different styles of "coding" the results. These include copy with substitution, generation, and inheritance.

As described herein above, objects have properties, interfaces and behaviors. Any property of any object instance can have two types of values. The two types are an explicit value and a derived value.

An explicit value is on set by a user and remains unchanged until reset by the user. The following rules apply:

1) If a property has an explicit value, then the explicit value is the one used for all purposes.

2) In the interface explicit values are denoted by being presented in bold text.

3) The interface for string explicit values allows for setting the value to
Null/Missing, so that a derived value can be used.

A derived value is calculated by an algorithm defined as part of the
object class. The following rules apply:

1). Derived values are calculated and recalculated on demand,
therefore the derived value for a property may change if any of the
information used in the algorithm changes.

2). Derived values are only used when the object property instance has
no explicit value set.

Derived values are presented in the UI as plain text. Derived value
algorithms can theoretically use any information in any way, but practically
use a number of frequently used styles, cliches, or categories.

These styles are: simple defaulting, intelligent defaulting, inheritance,
inheritance plus defaulting, complex derivation, and inheritance with
specialization.

With simple defaulting, the algorithm delivers a single value which is
always the same. Example, Property: Blank when Zero

Algorithm: Derived value is false

With intelligent defaulting, the algorithm uses property values from the
same object. Example, Property: Element label

Algorithm: Derived value is the Me.name, but
with _&-_ changed to spaces, and with each word starting with a capital,
followed by lowercase letters.

With inheritance, the algorithm copies the current values of a property
with the same name from an object that this object inherits from.

Example, Property: Element size

Algorithm: If Me.Inherits_From, is not null, then derived value
is Find_Object (Me.Inherits_From).Element_Size. This type of derived value
is widely used for event handler functions, where their syntax can be seen as
simply a long string property of the object.

With inheritance plus defaulting, if the object inherits from another
object, then inherit the property, otherwise default it.
Example, Property: element Picture

Algorithm: If Me.Inherits From, is not null, then derived value
is Find_Object (Me.Inherits From).Element_Picture. If fme.element_type is
numeric, then the derived value is a string of "^^" as long as Element_Size
with ",," before each third"^^".

With complex derivation, the algorithm uses information from several
sources, including Inherits From object properties, the resulting value is an
indirect reflection of the information used, not just a copy.

Example, Property: Column Datatype

Algorithm: If Find_Object(Me.Inherits From).element_type is
numeric AND me.parent.database_type is RMS AND me.used_in_indexes_list
is null AND Find_Object(Me.InheritsFrom).Element_Size < 10, then the
derived value is "integer" etc.

With inheritance with specialization, describes a variation on
inheritance, where the value (typically an event handler function syntax
string) can be inherited from another object, but then decorated with additional
information. The intended result is that the function can do more than the
inherited syntax, but still leverages off the inherited syntax in some cases or to
do part of its processing. To qualify as Inheritance with specialization, it is
important that the logic of the function does change automatically when the
logic if the inherited function changes. For example, the specialized function
could look like the following:

function event1 (a,,b,c)
    If <whatever>
      {do_inherited_event1(a,b,c);}  
    else
      {<specialized processing>}

Inherited From- Objects can be inherited from other objects for the
purposes of inheriting, or more generally, deriving property values from
them. Inherited from properties are strings rather than object pointers, the
object is found by searching for an object of the right name & type. The
search path is defined by the hierarchy of Library objects on the Repository diagram.

Referring to Fig. 30 there is illustrated in a Booch diagram normal inheritance. One object instance (A) inherits from another object instance (B). This means that the property values of A default to the equivalent property values of B. In this situation either object can be deleted without destroying the other. Deleting object A has no effect on object B, while deleting object B leaves object A with nothing to inherit from, so its default properties revert to the class defaults.

Referring to Fig. 31 there is illustrated in a Booch diagram list inheritance. One again object instance (A) inherits from another object instance (B), and object B contains a list of sub-objects (C1 to Cn). In this case Object A is given an equivalent list of sub-objects (D1 to Dn). This means that number and order as well as property values the list of D objects are derived from the list of C objects.

Referring to Fig. 32 there is illustrated in a Booch diagram changing the inherited object list of Fig. 31. Deleting a C object automatically causes an equivalent change in the D object list. For example, deleting Ci causes Di to be deleted.

Referring to Fig. 33 there is illustrated in a Booch diagram changing the inherited parent object Fig. 31. Changing the B instance that the A instance inherits from changes which list of C instances is used and therefore changes the list of D objects. For example, changing the parent object from B1 to B2 changes the C object list from (C1 to Cn) to (Cm to Cz) thereby effecting the D objects to become Dm to Dz.

Referring to Fig. 34 there is illustrated in a Booch diagram extending the inherited list of Fig. 31. Object A cannot delete objects from the inherited list of D objects, but it can add more objects to the list. For example, extending the D object list from (D1 to Dn) by adding (Dn+1 to Dn+m) thereby effecting the D objects to become (D1 to Dn+m).

Referring to Fig. 35 there is illustrated in a Booch diagram normal
inheritance with overriding of property values. One object instance (A) inherits from another object instance (B) as in Fig. 31, however one of the list objects has a property value is overridden. This means that the property values of A that normally default to the equivalent property values of B have been specialized by overriding one value. In this case "top" has been given a value of 300. The override value acts as an explicit value, the other property values are derived.

Libraries are container objects of reusable objects. Consequently, rules for inheritance between Application objects and Library objects are required and are as follows:

Objects within an Application can inherit from objects within a Library;

Objects within an Application cannot inherit from other objects in Applications;

Objects within a Library can inherit from objects in the same library or other libraries (within search list);

Objects within a Library cannot inherit from other objects in Applications.

Objects have an associated list of procedures. When one object inherits from another object, it inherits the list of procedures. A user can then choose to specialize (that is redefine) or inherit (leave as default) these procedures on a procedure-by-procedure basis. For example, the user may specialize the "Mouse_Down" procedure, but continue to inherit the "Click" procedure. Most objects have a finite and predetermined number of associated procedures. For example, a "button" object has MouseOver, MouseClick and a few others. These procedures have a predetermined names and are automatically triggered by events or methods on the object. Forms and modules can also have additional procedures that may be of type SUB or FUNCTION. The number and names of these procedures are entirely at the user's discretion. Inheritance and specialization apply to both of these procedure types.

The association between a procedure, the object it relates to and the event/method that it invokes is based upon the object name. For example, a
button called btnOK, would have associated event handlers called btn_OKMouseOver, btnOKMouseClick.

An object that inherits from btnOK needs to inherit the text of these procedures, but will have another name, e.g., btnOKinDialog1. Consequently the inherited procedures will be given the same text but different names, e.g. btnOKinDialog1_MouseOver, btnOKinDialog1_MouseClick.

Specialization of an inherited procedure involves replacing the text of the inherited procedure with some other text.

An example of applying inheritance is provided as an add to understanding. A user requires that all programs start the same way.

1) A splash screen appears with the name of the application, copyright, and bit maps.

2) The splash screen disappears after a user click or a time out, and is replaced by a security form.

3) The security form asks for username and password, and includes some trap doors for developers.

4) Once the username and password has been validated, the specific application begins.

To do this, the user would define, in a library, a presentation module called UI Startup containing a splash screen and a security form together with their associated logic. To achieve a uniform startup, each program created would inherit from UIStartup.

For the splash screen, the bitmap displayed would have to be specialized as a property of a Picture control, but the objects and logic can be just inherited. For the security form, most of the logic can be inherited. The areas requiring specialization would be calling the actual application, application identification, and user verification.

Referring to Fig. 36, there is illustrated, in a flow chart, a method of application program building in accordance with an embodiment of the present invention. To prepare for program building, using the workbench interface, a workspace and a repository are created. The workspace is a file in which
program objects are created. The repository is a place in which to store the completed work. From an existing database, data definitions are imported into the workspace. Once these preliminary steps have been completed, the program can be built through a series of simple steps.

First, an application object is selected in an explorer window of the workbench interface, as represented by a block 250. Next, from an insert menu 252, a program is selected. In response to the selection, a screen wizard prompts for the template desired, as represented by a block 254. The application program builder provides three different types of templates. The choice of template determines the kind of program being built. Once selected, a sequence of wizard panels are provided as indicated by blocks 256, 258, 260 and 262. The block 256 represents presentation of a data access panel for defining tables and relationships for the program being built. The block 258 represents presentation of a style techniques panel that determines how forms generated by the program will appear and behave. Defaults provide a basic style, as predetermined by the template selected, that can be customized. The block 260 represents presentation of an inheritance properties panel. The inheritance properties are used to indicate what parts of the program being built are based upon existing objects (e.g. library or repository). The block 262 represents presentation of a forms control panel. The forms control panel is used to indicate which fields, from each record set are used in the form created by the program being built. Form control panels appear for each record set identified in the data access panel. After a last form control panel has appeared, a set names panel appears, allowing naming of the program being built and its presentation, application, and data connection modules.

A dialog box appears with options to edit, build, run, and explore high-level objects of the program. Selecting build, as represented by a block 264, with the program object in focus, causes the program to be built, and provides results with errors and warnings for each high-level object, once completed.

A return to the explorer window allows the program to be run by
selecting run from the tools menu, as represented by a block 266. Alternatively, the dialog box run option could be selected, causing the program to be run.

Referring to Figs. 37-42, there is illustrated, in block diagrams, the user interface of the workbench of Fig. 1. In Fig. 37, a UI 310 requires a supporting data access services 312. The UI 310 includes a UI framework 314 and a UI API 316. The data access services 312 include a common object cache 318, a data server API 320, and data server implementations 322.

The Data and the User Interface are strictly insulated from each other. The UI’s only access to data is through the Data Server API 320. The UI neither knows nor cares about how (or where) the information is stored. The Data Server uses the UI API 316 to keep the UI informed about changes to its data; it neither knows nor cares how the UI is implemented.

The common object cache (COC) 318 exists to allow access to objects stored in different ways. The data server API 320 is a set of routines that the UI framework 314 uses to manipulate COC objects.

The UI architecture has four structural components:

The Common Data Server 320 defines the Common Data Access API and supports access to object stored using different storage mechanisms transparently by indirecting to the appropriate Implementation DLL. Also supports the Common Object Cache.

Data Server Implementation 322 is responsible for providing access to object data stored in a particular paradigm. Links to and uses the UI API DLL to keep the UI Framework informed of changes in the object data.

The UI API 316 is called by the Data Server Implementation DLLs to inform the framework about changes to an object’s data. The framework uses these routines to keep the Views synchronized with the current state of an object.

The UI Framework 314 uses the MFC supplied framework and extends it to provide the support necessary to allow multiple Views within a single.
MDI frame window. Data access is provided by implementing "Views" onto objects stored in the common Object Cache. It uses the Common Data Server API for all object data access. The Framework has support for multiple Views within a single MDI frame as well as providing the COC View "Abstract" class which greatly reduces the effort required to implement new View types.

In Fig. 38, a UI framework class hierarchy is shown. The framework hierarchy 330 includes standard MFS framework 332 and a framework extensions 334.

A Framework is the shell within which an application's functionality is implemented and presented. It provides the mechanisms and services which support the application without actually having to have in-depth knowledge of the application's domain. This shell is seen in such things as MDI applications in which the main windows frame keeps track of what MDI child windows are open and keeps the "Windows" menu list up-to-date. The MFC frame has extended support in the form of multiple toolbars, menus and status line definitions which can be linked to a particular MDI child. The framework 314 takes the standard MFC model and extends it with as few changes as possible to provide additional functionality specifically targeted towards business application program building.

Fig. 38 illustrates how the UI Framework 314 extends the regular MFC framework by introducing two new classes, a View container 338. The introduction of these classes is necessary because the default framework is not set up to handle multiple views in a single MDI frame. The rest of the description about the framework will be broken into three parts:

A description of the "View Container" class 336.

A description of the "COC View" abstract class 338 and its derivatives.

What modifications were necessary to the standard framework.

Containers are ordered lists of "Panes". A "Pane" can either be View or it can be another Container (nested groups). There are currently three
container styles supported:

- TAB
  Used when only one of the panes needs to be visible at a time
- SPLITTER
  Used when more than one of the panes needs to be visible simultaneously.
- WORKSPACE
  Used to allow "formatted" sets of Views with a defined layout.

Since the purpose of the container is to handle "Groups" of associated Views the implementation focuses its attention on the "standard" Group operations:

- Add - Add a new pane to an existing Container. This is accomplished simply by creating the new View/Container as a child of the Container you want it added to. The Container class detects the window's creation and automatically adds it to its list of panes.
- Delete - Remove a Pane from a Container. This is accomplished simply by destroying the view/Container window. The Container class detects the window's destruction and automatically removes it from its list of panes.
- Copy - Making a copy of an existing Pane and adding it to another Container. The Container and View classes must each supply a MakeACopy method which is responsible for being able to clone itself when given the new parent Container as a parameter. If the pane being cloned is a Container then all internal Views and Containers are recursively cloned.
- Merge - Move an existing pane into a new Container. This is actually implemented by performing a "Clone" operation into the new Container and then deleting the old one (although Windows provides a SetParent function which should allow us to simply move the window under the new Container the
implementation of this operation has bugs which prevent us from using it).

- Extract - Break an existing pane out into a new MDI Frame. This is actually implemented by creating the new MDI Frame and then "Merging" the existing pane into it.

An MDI frame now contains a "Main" Container which becomes the "root" of a tree of windows in which the branches are other Containers and the leaves are COC Views. The standard framework still keeps track of which MDI Frames are open ... while the Container class provides the grouping operations and internal control, e.g., which View is showing in a Tab, how wide the Splitter panes are. The Containers are also aware of the COC in that they have their own "Selected ID" which all of their children set whenever a new object is selected by the user. This is necessary to allow other Views to modify the object they're viewing based on a change in the selection state in ANY of the Views in a particular container.

Referring to Fig. 39, there is illustrated for a running UI Framework application. The arrows represent selection "links" which will cause one View to modify which object it's showing based on the selected object in another pane at (1) the Property View will show the last object selected in ANY of the Views to the left of the Splitter. At (2) the right List View will show the children of the last object selected in EITHER the left list or Diagram.

Views are by far the most important single piece of the framework.

- All Views are onto Objects that exist in the Common Object Cache. This allows the view to be shared across objects using different storage mechanisms (e.g. a List could either show an Application's Forms or a Form's fields).

- Views should not locally store object information. The cache_info struct in the COC is the preferred way to maintain data for quick access. Storing object data in the View would require that the View be able to keep itself synchronized correctly with the actual object's state; while possible, this is
seen (in most cases) to unnecessarily complicate the View's implementation. Information about an object should be retrieved through its Properties.

The functions that a View must be able to perform fall into two categories:
- Functions that are shared between ALL Views
- Functions that are specific to a particular View

C++ provides a paradigm for describing this situation called an "Abstract Class". The API necessary to support ANY View is described, default functions provided for those routines that are common and explicitly declaring the API's for those functions that are individual View needs to provide.

Any routines that are implemented in the Abstract Class do not have to be re-implemented in the derived Views. Explicitly defining the "required" routines allows the shared routines to make calls to routines that are implemented in a particular View without knowing what that View is. Almost all Views can be implemented by providing SIX functions (aside from any necessary internal routines):
- Clear - moves the previous object’s information from the View
- Populate - sets the View up to show a new Object
- AddObj, DeleteObj & Refresh - used to keep the View synchronized with the database
- ObjFromPoint - a "Pick" function (used by the shared Drag and Drop handler).

Using the Abstract Class greatly reduces the number of routines required to implement a new View. Also, since the routines implemented in the Abstract Class are used by all Views, enhancing them automatically means enhancing all the derived classes. The more complex the process involved, the more advantage that is gained by implementing it in the Abstract class.

This can be seen in the current implementation by observing that all of the synchronization handling is implemented in the Abstract class. The flexibility
is maintained because the routines implemented in the Abstract Class can be "over-ridden" by providing a function with the same name and API in the actual class.

As stated earlier there are a minimum number of changes to the framework provided by the MFC. The changes are:

- Overriding OnFileNew and OnFileOpen for the App. The standard routines provided by the framework expect that the files are in a format particular to the MFC's Document/View architecture ("Serialized").

- Modifying the OnActivate function for MDI frames to inhibit their Menu and Toolbar behaviour. The extended architecture changes menus, when a View becomes active, not the frame it resides in.

- Two modifications to get around restrictions in the standard Document/View implementation. In the MFC framework new views are created by suing a Document "Template" which controls things like the MDI Frame's style and the type of View that gets created inside it. In order to allow multiple Views within a single frame we had to separate ourselves from the template mechanism. This was accomplished by defining a "Standard" template which creates the frame and populates it with a "fake" view (the fake is necessary because the MFC framework requires that all frames contain at least one view at creation time). The second was to store a pointer to the document's Context in the MDI frame. This "Context" allows the document to keep track of all Views created onto it.

The purpose of the Data Server is to isolate access to the data used by the UI behind a formalized API. The advantage of this is to minimize the effect of changes in the underlying data models (or, indeed, in the object storage mechanisms) on the UI implementation.

The Data Server implementation must provide common access to
objects stored in different data storage paradigms. This is necessary to provide the "integrated" feel to the environment on the UI side; the Views are the same regardless of the type of the objects they are looking at. Another thing that it should provide is a mechanism which allows the implementor of a particular Data Server to locally "cache" information about an object to improve performance.

The data Server implementation is made up of four major components:

- The Common Object Cache ("COC") 318
- The Common Data Server DLL ("DS") 320
- a number of Data Server Implementation DLLs 322
- The UI API 316

All of the components of the Data Server are implemented in regular "c" code. This is a reflection of the fact that the currently used data sources are implemented in "c" and also because there is a standard "c" calling protocol*. Using this protocol keeps the Data Server as open as possible since it would be able to service clients written in either "c" or in C++.

*C++ "decorates" (mangles) function names to allow overloading.

Unfortunately, there are currently no standards as to how this decoration takes place so it is difficult to call C++ defined routines from "c".

The common object cache is illustrated in Fig. 40.

The first component of the Data Server implementation is the Common Object Cache 318. The COC 318 provides the common "mask" behind which the actual objects reside, thus allowing the UI to remain ignorant of an object's particular storage mechanism. The Data Server API 320 uses the COC "ID" 350 whenever referencing an object.

The amount of information stored in the COC is intentionally kept to a minimum to avoid overhead (remember, all objects represented in the UI must have a COC counterpart). There are only three essential pieces of information that must be stored:

- The "ds_type" of the object; used to map the common DS API call to a particular implementation. How this mechanism works
will be explained in the discussion on implementing a Data Server later in this section.

- The "db_id" 354 that is the id of the actual object in its own data space
- The "cache_info" that can be sued to locally cache extra information about an object

In addition to this info, the COC also maintains information about the hierarchy of all the objects it contains. While not directly exposed to the Data Server implementations 322 this allows the Cache API to provide iteration and search functions across the COC. This is used in the DS implementations to "look-up" the COC ID of an object when given only the "real" db_id. There is also a statically defined "root" object which defines the top of the COC hierarchy.

The common Data Server DLL 320 has a main function to provide the UI Framework 314 with a formal API with which to perform all data access. It must act as a bridge between the UI 310 and the actual Data Server Implementation 322. This is accomplished through using the Common Object Cache (for data) 318 and a Data Server Function Table (for operations). It exports two distinct sets of routines;

- to the Data Server Implementations it exports the Common Object Cache manipulation routines and its own "registration" routine. The COC API consists of routines that allow DS Implementations to add and delete objects from the cache as well as query routines that can be used for such things as getting a list of the children of a given object or finding an object in a cache from its "real" data source IC. The Data Server Implementations use this API to keep the COC in sync with the objects actually stored in the true database.
- to the UI Framework it exports the Common Data Access API. The Common Data Server API represents the routines through which ALL data access is performed. It can be broken into
four functional areas:

- Data Server Control (Init/Term, Open/Close/New, ...)
- Object Control (Add/Delete/Copy, ...)
- Object Property Control (GetPropInfo, GetProp, SetProp, ...)
- Data Server Queries (OKToDrop, OKToLink, ...)

The UI Framework will use these routines to display and manipulate the objects represented in its Views.

Fig. 41 illustrates how registration is handled,

1. Each Data server Implementation 322 fills in the addresses of the routines that it has implemented into a DB Server Function Table. It then exports its registration function to the UI Framework 314.
2. The Framework calls the registration routine during startup.
3. The Server exports that address of its function table using the DBRegister routine.
4. The Common Data Server stores the address of the table in an array during registration and returns the array index as the "ds_type". (Using the array approach minimizes the call overhead by preventing a "look-up" step for each call.)

During use by the Framework the Data Server DLL 320 treats these routines as a "pass-through" call, as illustrated in Fig. 42.

1. The UI Framework 314 makes a call using the Common Data Access API.
2. The Data server type is determined from the parameters (either directly passed in or from the COC if the parameter list contains COC object references).
3. The Common Data Server checks the appropriate DS Implementation's function table for the routine.
4. The Common Data Server calls the function if it is defined.
5. ...And passes back the return value through the common API.

* If the function is not defined in the implementation the common API returns a default result (FALSE for booleans, 0 for counts and
Implementing a Data Server 322 consists of implementing some subset of the Common API and filling in a static table of these functions. The format of the table is provided by the Common Data Server header files. Since all the implemented functions are accessed through a DLL/DLL callback they are defined as "_export" in the implementation. The DLL also publicly exports an "initialization" routine which the Framework will call to allow access to objects stored in the particular storage format implemented in the server. This routine should perform any necessary internal initialization and, if successful, call the DBRegister function of the Common API to export its function table.

It is the responsibility of Data Server DLL to populate and maintain the COC in synch with the objects existing in its own data space. It is also its responsibility to use the UI API routines to keep the UI Framework informed of any object additions, deletions or property changes which take place to objects in its domain.

A Data Server Implementation can also decide to locally store object information by packaging this information in a structure and setting the COC object's "cache_info" pointer. If an implementation does this it is then its responsibility to insure that the local information is kept in synch with the actual data. Since all object internal data is accessed through the Property API, the Get and Set property routines must be able to identify which properties are locally cached and which must be requested from the Data Server itself.

The UI API 316 provides the formal definition of the routines that the Data Servers use to keep the UI Framework informed of changes to object data. Implementing the API using a DLL is required mainly due to a linkage peculiarity. The Data Server DLLs need to be able to "call back" into the UI Framework. Unfortunately, there is no mechanism that allows a .EXE to "export" routines to an external DLL. So the Data Server implementation DLLs is linked with the UI API DLL and during the application initialization
the addresses of the appropriate framework functions are exported to this DLL. The UI API DLL works the same way as the common Data Server, except that it only maintains a single instance of its function table.

Numerous modifications, variations, and adaptations may be made to the particular embodiments of the invention described above without departing from the scope of the invention, which is defined in the claims.
CLAIMS:

1. An application program builder comprising:
   a plurality of predetermined high-level objects;
   a plurality of predetermined templates for building corresponding
   programs based upon the high-level objects; and
   a workbench interface for providing access to and manipulation of the
   templates and high-level objects.

2. An application program building as claimed in claim 1 wherein
   the plurality of predetermined high-level objects includes an application object,
   a database object and a library object.

3. An application program building as claimed in claim 2 wherein
   the application object includes a program object.

4. An application program building as claimed in claim 3 wherein
   the program object includes a presentation module object, an application
   module object and a data connection module object.

5. An application program building as claimed in claim 4 wherein
   the application module object includes a data access model object, a data
   transaction object and a record set object.

6. An application program building as claimed in claim 5 wherein
   the data access model object includes a linkage object.

7. An application program building as claimed in claim 5 wherein
   the record set object includes a field object.

8. An application program building as claimed in claim 4 wherein
the presentation module object includes a form object.

9. An application program building as claimed in claim 8 wherein the form object includes a control object.

10. An application program building as claimed in claim 8 wherein the form object includes a data control object.

11. An application program building as claimed in claim 4 wherein the presentation module object includes a toolbox object.

12. An application program building as claimed in claim 4 wherein the presentation module object includes a menu object.

13. An application program building as claimed in claim 4 wherein the presentation module object includes a statusline object.

14. An application program building as claimed in claim 4 wherein the data connection module object includes a data connection object.

15. An application program building as claimed in claim 1 wherein the plurality of predetermined templates includes a master-detail template.

16. An application program building as claimed in claim 15 wherein the master-detail template creates a form having a common header information and rows of detail information below the common header information.

17. An application program building as claimed in claim 1 wherein the plurality of predetermined templates includes a multi-form template.

18. An application program building as claimed in claim 17 wherein
the multi-form template creates multiple forms to represent multiple tables in an application.

19. An application program building as claimed in claim 1 wherein the plurality of predetermined templates includes a reference list template.

20. An application program building as claimed in claim 19 wherein the reference list template creates a drop-list on an existing form for providing access to a reference table.

21. An application program building as claimed in claim 1 wherein the workbench interface includes a user interface and data services.

22. An application program building as claimed in claim 21 wherein the user interface includes a user interface framework and a user interface application program interface (API).

23. An application program building as claimed in claim 22 wherein the user interface framework is an MFC framework with extensions.

24. An application program building as claimed in claim 23 wherein the extensions include a view container and a common object cache view.

25. An application program building as claimed in claim 24 wherein the view container is a tab.

26. An application program building as claimed in claim 24 wherein the view container is a splitter.

27. An application program building as claimed in claim 24 wherein the view container is a workspace.
28. An application program building as claimed in claim 22 wherein the user interface API includes routines for use by data services to inform the UI framework of changes to object data.

29. An application program building as claimed in claim 21 wherein data services include a data server API, data server implementations and a common object cache.

30. An application program building as claimed in claim 29 wherein the common object cache includes a common object cache identifier for every object in the user interface.
Fig. 3

Fig. 4

DATABASE

TABLE

VIEW
FIG. 12

FIG. 13
FIG. 14

FIG. 15
Fig. 18
FIG. 21

11/22

Customers
Customer No. [Box]
Name [Box]

Details

Customers Details
Orders
Sites

Customers
Customer No. 10
Name [Box]
Next

Changes are synchronized

Customers Details
Customer No. 10
Orders | Sites

FIG. 22

SUBSTITUTE SHEET (RULE 26)
12/22

FIG. 23

FIG. 24

SUBSTITUTE SHEET (RULE 26)
FIG. 29

FIG. 30
Fig. 33
Fig. 34

Fig. 35
Fig. 36

Substitute Sheet (Rule 26)
FIG. 41

FIG. 42
INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>IBM TECHNICAL DISCLOSURE BULLETIN, vol. 33, no. 11, 1 April 1991, pages 184-191, XP000110362 &quot;GRAPHICAL PROCEDURAL CAPABILITY&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

[X] Further documents are listed in the continuation of box C. [ ] Patent family members are listed in annex.

* Special categories of cited documents:
- "A" document defining the general state of the art which is not considered to be of particular relevance
- "E" earlier document but published on or after the international filing date
- "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
- "O" document referring to an oral disclosure, use, exhibition or other means
- "P" document published prior to the international filing date but later than the priority date claimed

"I" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search
10 September 1997

Date of mailing of the international search report
2 3. 09.97

Name and mailing address of the ISA
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk
Tel. (+ 31-70) 340-2040, Fax: 31 631 epo nl

Authorized officer
Fonderson, A
<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY, vol. 34, no. 10, 1 October 1991, pages 64-77, XP000266967 BUTTERWORTH P ET AL: &quot;THE GEMSTONE OBJECT DATABASE MANAGEMENT SYSTEM&quot; see page 70, left-hand column, paragraph 2 - page 76, right-hand column, paragraph 1; figures 5-7</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>SIGMOD RECORD, vol. 22, no. 4, 1 December 1993, pages 11-18, XP00013415 ARPINAR I B ET AL: &quot;MLOODVIEW: AN ADVANCED GRAPHICAL USER INTERFACE FOR OODBMS&quot; see the whole document</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>IBM TECHNICAL DISCLOSURE BULLETIN, vol. 37, no. 2B, 1 February 1994, pages 73-77, XP000433766 &quot;VIEW EDITOR&quot; see the whole document</td>
<td>1</td>
</tr>
</tbody>
</table>

Form PCT/ISA/210 (continuation of second sheet) (July 1992)