Accessing a Medical Database using WWW-Based User Interfaces[1]

Saurabh Sinha, S. Kirk Bowers, and Sandra A. Mamrak
Department of Computer and Information Science
2015 Neil Avenue Mall
The Ohio State University
Columbus, OH 43210
{sinha,bowers,mamrak}@cis.ohio-state.edu

March 16, 1998

[1]This work was supported in part by Grant No. 5 U01 CA50910-08 from the National Cancer Institute.

Abstract

We have implemented a Neuro-Oncolgy Information System (NOIS) for researchers and other personnel engaged in brain-tumor research throughout the United States. In this paper we describe our migration of the implementation of the NOIS user interface from Oracle-based products to Java applets. Our goal in the migration was to design the interface software not only as a set of reusable objects, but also to reuse the patterns of interaction among the components by way of object-oriented frameworks. We describe in detail the design of the user interface classes and the connection objects that let Java applets communicate with the database, and we evaluate our success in meeting our design goal.

Keywords and Phrases: brain-tumor research, graphical user interface design, object-oriented design, software reuse

1 Introduction

Software reuse has a significant potential for reducing both development and debugging time for new applications, sometimes by as much as an order of magnitude. These reductions, in turn, lower the cost of product development and increase the quality of the new applications.

Many technical and nontechnical factors must coalesce in a software development project to realize the potential that reuse provides. Chief among the technical factors is a software project whose inherent architecture is able to be generalized and abstracted into a set of components that are commonly used by applications. Considerably more advantage can be realized if the patterns of interaction among the components are similar or identical across different applications in the project.

Many nontechnical factors can mitigate against success. An emerging consensus in the reuse community [1] is that the primary nontechnical reuse barriers stem from the organization and management of the software development team and include:

- A poor investment in organizational learning.
- A lack of attention or effort to develop a complete software architecture, including not only tangible products (languages, databases, class libraries, etc.), but also methodologies and guidelines needed to effectively use those products.
- An attempt to address too many changes in a software development practice, i.e., changing in one iteration to an OO programming paradigm, to a framework approach and to an emphasis on systematic reuse.
- An organizational lack of incentive to build for reuse.

In this paper we describe a case study of a software development project in which the technical and nontechnical factors affecting reuse aligned themselves in the most propitious circumstance possible, allowing close to total code reuse in the generation of new applications in the project.

2 Background

We have been serving the data storage and retrieval needs of a national group of brain-tumor researchers for many years by way of an information system designed for this group of collaborators [2, 3]. The Neuro-Oncology Information System (NOIS) follows a client-server architecture. The server is implemented as a centralized database and the clients run on heterogeneous platforms scattered throughout the United States.

The NOIS database server is implemented using Oracle relational database software [4]. Originally, the client software also was based on Oracle software: the Oracle Data Browser [5] for query capability and Oracle Forms [6, 7] for input capability. We have now migrated our client software to WWW-based Java code.

2.1 The Neuro-Oncology Information System

The NOIS is a collaborative project established at the Ohio State University with the goal of supporting cooperative research in brain tumors. To date, the NOIS supports two specific groups of researchers: (1) The Brain Tumor Program Project, and (2) The Glioma Marker Network. Generally speaking, the goal of both projects is to improve the diagnosis and treatment of brain tumors through the collaboration of a team of researchers. The projects emphasize cooperative exchange of ideas and resources to achieve their goals. NOIS supports this exchange by providing an information system to store, retrieve and integrate clinical and tissue-inventory data for both groups.

The NOIS database stores information from both the Brain Tumor Program Project and the Glioma Marker Network in a unified schema. The different kinds of information in the NOIS can broadly be categorized as:

- An *inventory* of tissue specimens and related data. Tissue is obtained by clinical activity and is related to various clinical data.

- A variety of *clinical* data about patients and their treatments for brain tumors. Information about treatments include subsystems for surgery, radiotherapy, chemotherapy and non-treatment interactions. Other data relating to patients includes demographic and administrative data.
- *Research results* and other research-related data that is to be shared.

2.2 User Interface Forms in NOIS

The users of NOIS interact with the database for input and query purposes through user-interface forms — the NOIS Input Forms support input capabilities, and the NOIS Browser supports query capabilities. User interface plays an important role of abstraction between the users and the database. It disassociates the users from the intricate details of the relations in the underlying schema, and presents them with a logical, coherent view of the data that can be easily understood. In NOIS, a number of user-interface forms are provided corresponding to the class of users and the data stored in the system. This further reduces the complexity from the users' point of view, because each user group only sees information that is of direct interest to itself. There is only one NOIS Browser, although any number of different queries can be executed in the Browser. The different types of Input Forms in NOIS are:

- Inventory. Used to input data to the inventory subsystem of the database.
- Clinical_Patient. Used to input data to the clinical subsystem of the database when patient-identifying data are available.
- Clinical_Tumor. Used to input data to the clinical subsystem of the database when the data is linked to a tumor and not a patient. This occurs when a tumor is obtained from a National Tumor Bank and patient-identifying data is purposely suppressed. The clinical data are also provided in a form different from that used in the Clinical_Patient form.
- Radiotherapy. A restricted version of the Clinical_Patient form that includes only patient and radiotherapy information.
- Set_Case. Used by a clinical input person to link the clinical subsystems associated with a patient with the inventory subsystem associated with a tumor obtained in a surgery on the patient.
- Research Input Forms. Used by individual researchers who use the NOIS to store their research data.

2.3 Migration from Oracle Forms to Java

The user interfaces were initially implemented using the Oracle Forms Designer. This tool provides a layout editor for building user interface screens by creating canvases of interface objects like text item, button, checkbox, list of values, etc. The Oracle Forms Runtime has to be installed on each client machine that wishes to access the database through the user interfaces. The connection between the Forms interface and the database is handled transparently through Oracle's proprietary communication protocol called SQL*Net [8]. Versions 1 and 2 of the NOIS Input Forms were implemented in this manner.

In the course of maintaining the Input Forms we experienced the classical problems related to supporting client-side software:

- multiple versions: because not all clients update their versions when a newer one is available, we were required to maintain multiple versions of the client software;
- platform-dependent copies: because our clients were using Macs, PCs and Unix-based workstations, we had to maintain client software for each platform, in multiple versions;
- remote installation/maintenance: users often did not have sufficient support staff to initially install client-side software, or to upgrade it when necessary, and would request this service from us;
- firewalls: almost all of our clients, who work in medical research labs, are behind firewalls and standard versions of commercial client software cannot penetrate most firewalls.

We also experienced the classical problems that arise when generating similar applications in a project, with little effort to maximize reuse potential. The development of each form was a relatively lengthy process subject to considerable error, depending upon the knowledge and skill of the individual programmer that was in charge of each form. Maintenance was also problematical when the original programmer left and others had to modify or correct the code.

Oracle is phasing out its support of its Forms products and we were faced with a migration of our Oracle-based implementation to a different implementation. We took this opportunity to enumerate the desirable characteristics of the new implementation. The forms were quite complicated in terms of the look-and-feel, navigation among various screens, and connection to the database. Thus, we desired a solution that would allow us to manage this complexity both in the design and development phases as well as in maintenance. Reuse was the highest priority and we looked to an object-oriented paradigm as the way to achieve it.

Based on our experience with client-side software installation and maintenance, we also desired a solution that allowed for developing and maintaining only one, machine-independent version of each form, that did not require client-side installation, and that could penetrate firewalls.

The Java paradigm [9, 10] presented to us a truly compelling, revolutionary solution, having all of the desired characteristics for which we were looking. Java is object-oriented, so could help us manage complexity both in terms of high-level design methods and its reuse capability.

Java platform-independent applets reside on a server and are downloaded to the client through a World Wide Web browser when an application is invoked. Maintenance would be greatly reduced because we would be able to maintain only one version of one copy. There is no remote installation requirement (aside from having a WWW browser) and Java applets penetrate most firewalls.

We are aware of no other approach to, or toolkit for, user interface design that meets all of our desired criteria. For example, Tk/TCL [11] is a scripting language, and not able to easily handle the level of complexity of our forms or to easily support reuse. The X Window System/Motif [12] is tied to Unix platforms only. The Application Environment Specification from the Open Software Foundation [13] must be installed on the client. Commercial database forms systems typically will not penetrate firewalls (e.g., this is true for Oracle Forms).

2.4 Factors Affecting Reuse in the NOIS Project

Ours is relatively small development project, set in an academic Computer Science department. The technical and management leadership for the team is provided by a Professor in the department. Computer Science graduate students comprise the development team. Four to five students are supported to devote approximately twenty hours per week to the project. Typically between five and ten more students are associated more loosely with the project, through research seminars and independent-study courses. These students devote from six to ten hours per week on the project.

2.4.1 Nontechnical Factors Supporting Reuse

In this environment, we are able to overcome easily the primary nontechnical barriers to effective reuse:

- Obviously, learning is a primary consideration in an academic setting. A continued investment by way of formal and informal teaching methods is a constant in the project.
- In the face of relatively high turn-over of students participating in the project, considerable effort and attention is continually devoted to developing a complete software architecture. (See URL http://acuity.cis.ohio-state.edu:8888 for documentation of our architecture.)
- The project has been developing over many years and development changes have been steadily introduced over the time period.
- The motivation among team members is extremely high to participate in leading-edge software development, such as that dedicated to successful reuse, in preparation for future careers.

2.4.2 Technical Factors Supporting Reuse

When we first started our new, object-oriented design of the Input Forms, we were able to identify quickly and easily a set of objects that were common to all the forms. In this sense our project was such that its "inherent architecture is able to be generalized and abstracted into a set of components that are commonly used by applications." We eventually were also able to identify, and encode, common patterns of interactions among the components that were essentially identical across different applications in the project.

Common Components. Even though among the various forms there were more than twenty different screens, displaying more than a hundred data fields from tens of database tables, we could identify areas, or objects, common to all the screens: a menu bar, a context display area, a data-input portion, a table object (for displaying data retrieved from the database), a button row, and a message panel. Similarly, within a given area, we could identify easily objects such as menu items in the menu bar, data fields in the input portion of the screen, buttons in the button panel, and so on.

The design and implementation of these common objects, and their relationships in a class hierarchy, dominated our first Java implementation of both the NOIS Browser and the NOIS Input Forms. These first object-oriented versions provided the development team with (1) valuable experience in the object-oriented design process and, (2) an opportunity to learn and use Java on a decidedly non-trivial application.

These first object-oriented versions, however, were based on a fairly static view of objects and their class hierarchy. For example, although we had identified a button object, we had not distinguished, in the class hierarchy, *based on behavior*, an Add button from an Ok button or an input field to which a date was written from one to which free text was written.

Similarly, we did not specify or encode in any methodical and careful manner the way in which a particular object interacted with other objects, or consider whether the patterns of interaction were common to some or all objects of that type, regardless of on what screen they were instantiated. The extent of our reuse of patterns of interactions among objects was to cut-and-paste pieces of code from the implementation of one object to another, if and when we recognized this was possible.

While debugging these first-version forms we quickly realized that even though we gained some time in the development stage from input form to input form, we had gained little in terms of the time to debug a Java form over the time for its equivalent Oracle-based implementation.

Common Patterns. At this point in the project we advanced from using a simple object-oriented approach to one in which our goal was to create an object-oriented *framework* [14, 15]. Software developers have a variety of definitions and conceptualizations of frameworks. The description that is closest to our use of the term is that a framework is a reusable, not fully-specified application that can be specialized to produce custom applications. It embodies both design and implementation decisions in code (i.e., Java for our work), in a way that not only makes the objects reusable, but also makes the patterns of interaction among the objects reusable. Frameworks differ from other class libraries by reusing high-level design implicitly encoded as dependencies among classes.

The primary benefits of object-oriented frameworks stem from the modularity, reusability, extensibility and the inversion of control they provide. Inversion of control is realized when the framework is customized to generate specific applications. A specific application responds to external events by calling event-handler objects in the framework. Thus, the framework, as opposed to the customized application, controls which set of application-specific methods to invoke in response to external events.

The framework concept is ideally suited to our forms work because, for the Input Forms in particular, not only are many objects shared across specific forms, but the patterns of interaction among these components are essentially identical regardless of where the object is instantiated.

We applied the framework concept to both the new NOIS Data Browser and the new NOIS Input Forms. The interface for the Data Browser and Input Forms are quite different and overlap primarily in their use of an initial login capability and their access to the database. They are also different in the sense that while there is only one Data Browser, there are many different Input Forms. Thus, the design of the Input Forms was much more challenging and interesting than that for the Data Browser in terms of our reuse goals, and it is the one we emphasize in this manuscript.

3 User Interface Design

The NOIS Browser and the various NOIS Input Forms are executed from within a common overall system architecture. In this section, we first describe the overall architecture, and then present the design of the Browser and Input Forms.

Our emphasis in this manuscript is on the migration of the user interface components from their previous Oracle-based format to the new Java format. It is beyond our scope to address here human-computer interaction issues with respect to the interfaces, e.g., justify the design or present data on ease of use. We can observe that the NOIS Browser was modeled after the Oracle Data Browser, but is much simplified.

We found that the great majority of our users were not interested in or capable of dealing with the added complexity. The NOIS Input Forms were designed by John Boyd, who has many years of HCI experience in industry, as well as a recent Ph.D. in that area.

3.1 Overall Architecture

Figure 1 presents an overview of the architecture to support the NOIS WWW-based forms. It shows the major components of the system, and their interactions.

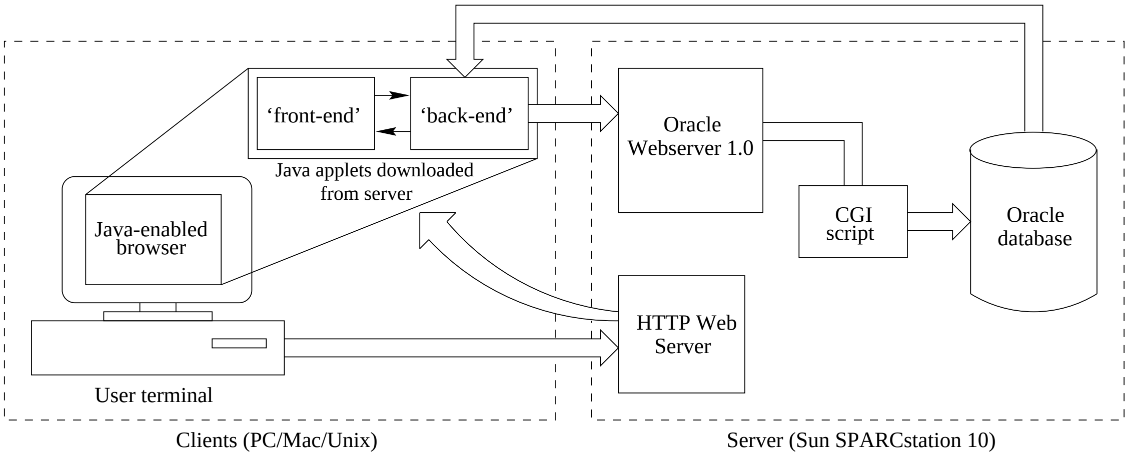

Figure 1: System architecture of the NOIS WWW-based forms

The server houses the database and runs two web servers: one of them is an HTTP server and the other is Oracle WebServer 1.0. Using a Java-enabled web browser, clients connect to the HTTP server and download Java applets from the server. These applets are comprised of a 'front-end' component and a 'back-end' component. The 'front-end' manages the interface with which users interact; the 'back-end' handles all database accesses. The 'back-end' connects to Oracle WebServer with appropriate parameters. The server in turn invokes a CGI [16] program that posts a user request to the database. The results from the database are interpreted by the 'back-end' and passed on to the 'front-end'.

The 'front-end' includes graphical user interfaces for the NOIS Browser and the Input Forms, through which users interact with the system. The interface allows users to issue commands to the system and view the results, and in case of the Input Forms, enter and modify data. Apart form the user interface, the 'front-end' also contains database objects that form a layer between the user interface and the 'back-end'. Figure 2 shows a magnified view of the 'front-end' component. The database objects control flow of data between the user interface and the 'back-end'. In doing so, they also transform data from its representation in the user interface to a form required by the 'back-end'.

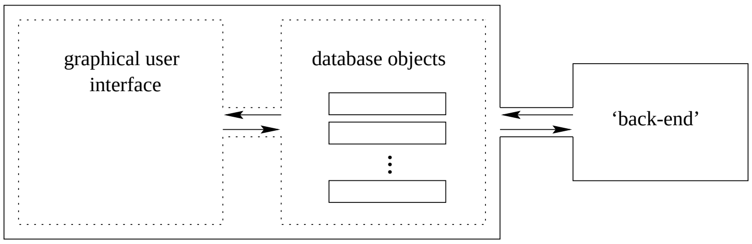

Figure 2: Detailed view of the 'front-end' component

3.2 The NOIS Browser

The NOIS Browser provides its users with an interactive method for constructing a query and limiting the scope of a query to the users' specific needs. In most respects, the interface design of the NOIS Browser is based on that of the Oracle Data Browser. Simplifications have been made so that users need not be familiar with the underlying database structure. The degree of flexibility for setting constraints has been reduced, allowing a more simplified method for setting conditions on a query.

The NOIS Browser is started when a Java-enabled web browser is directed to the NOIS Browser launcher URL. The NOIS Browser is a Java applet that opens windows onto the user's workdesk outside of the web browser window.

The first window that appears is a Login/Password window. All users of the NOIS Browser must be validated before use. Upon successful log in, the user is presented with a selection list window. The choices in the selection list are the queries to which that particular user has been granted access.

Once the user has selected a query, the Browser window appears (see Figure 3). The window consists logically of two halves. The two halves of the window map indirectly to the two separate windows of the Oracle Data Browser. The top half is referred to as the *builder* portion. The bottom half is the *results* portion. At the extreme bottom of the window is the button panel, from which the user is able to perform various actions.

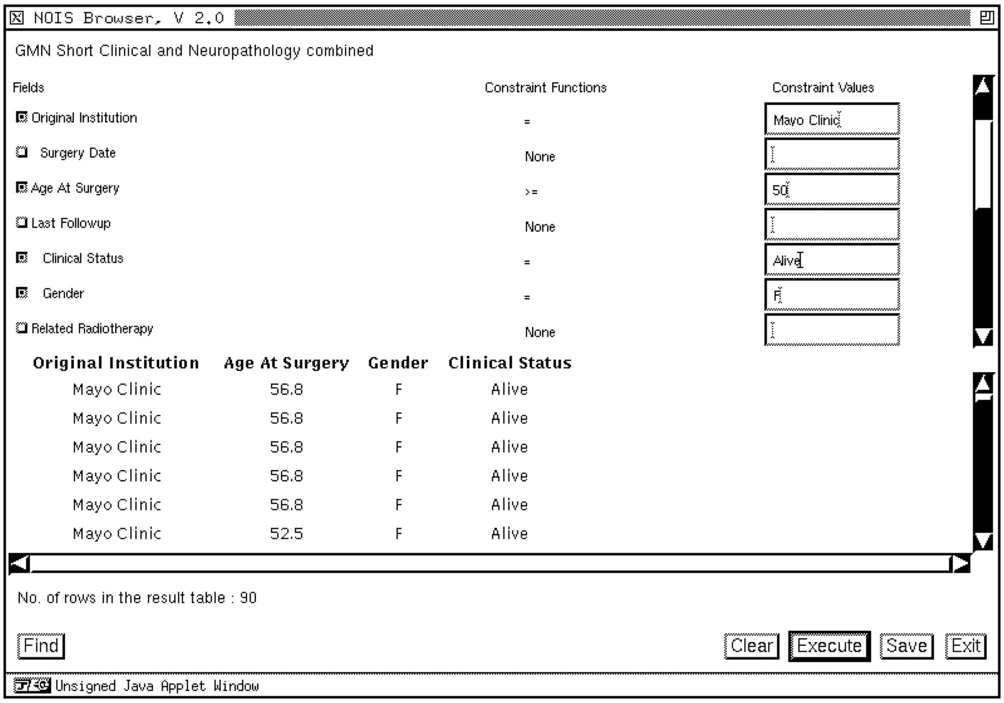

Figure 3: An example NOIS Browser window

3.2.1 The Builder Portion

The top line of the builder portion of the NOIS Browser window contains the title of the query. The remainder of the builder consists of a grid layout with three columns and as many rows as will fit. Each row corresponds to one data item, or field, in the query. The three columns are labeled "Fields", "Constraint Functions" and "Constraint Values." There is a scroll bar on the right in case there are more fields in the query than will fit on the screen.

The leftmost column, "Fields," is comprised of labeled check boxes. The labels are the names of the query fields. The check boxes indicate which of the many fields are to be included in the query. By default, all fields are excluded. It is up to the user to choose the values by selecting the check box next to those fields.

The center and rightmost column work in conjunction with each other to represent constraints set on the query. The middle column is comprised of pop-up choice menus. By default, all menus are set to the value ‘None’ to indicate that no restriction is being placed on that value. The other choices are the standard comparison operators (“=”, “!=”, “<”, “<=”, “>” and “>=”). The rightmost column is a free-form text field in which the user can specify the value to which that field should be compared. Initially, this text field is disabled until a constraint function other than ‘None’ has been set. The “Constraint Values” text field may be populated by the user typing a value directly into it, or by using the Find button described below.

In Figure 3, in the builder portion of the NOIS Browser, the fields ‘Original Institution’, ‘Age At Surgery’, ‘Clinical ‘Status’, and ‘Gender’ have been chosen to be included in the query. Each of these fields has been constrained to certain values, although in general a field may be included in a query and not be constrained.

3.2.2 The Results Portion

The results portion of the NOIS Browser window contains a spreadsheet-style table that displays the current state of a query. Just below the table is displayed a message stating the number of rows being held in the table. When a query is in the process of execution, the “number of rows” message goes blank as a visual cue that the number of rows is in the process of changing.

The results table interactively responds to user actions in the builder portion. When the check box by a field is checked, a column for that field appears as the rightmost column in the results table. Upon addition, there is no data for such a field, the data cells in a new column all contain blank values, even if other columns contain data from a previously executed query; data for the newly added field is fetched when the query is re-executed. When a field is unchecked in the builder, it is removed from the results table, and all columns that appeared to its right are slid down to fill the newly created gap. The order in which fields appear in the results portion can be rearranged by removing and re-appending the fields.

In Figure 3 ninety rows have been selected as a result of executing the query. From the results that are visible in the Figure, it can be seen that records were returned about alive, female patients from Mayo Clinic who were older that fifty years at the time of surgery.

3.2.3 The Button Panel

There are five buttons that appear at the extreme bottom of the Browser window. These buttons are Find, Clear, Execute, Save, and Quit. The Find button appears on the left, and the other buttons appear on the right. Find is separated because its function is unique. Also, this ensures consistency of look-and-feel with the Input Forms. The functionality served by each button is:

- *Find.* The Find button provides a selection list of all possible values for a particular field and may be used for setting Constraint Values. The Find feature is especially useful for constraining a field to equal a long name in which spelling, capitalization or punctuation will affect whether or not a match is found (e.g., “The Ohio State University Hospital”).

  In order to use the Find button, the context of the field for which to perform a find must first be specified. This is done by setting the “Constraint Functions” for that field to a function other than ‘None’, then clicking the mouse in the “Constraint Values” text field for that field to make the cursor in that text field active. Once the context is set, pressing Find will bring up one of two windows. If the context is a field for which find data is meaningful (hospital names, tumor numbers, gender, . . .), a selection list with all possible values is displayed. Once the user selects a value from this list, it will appear automatically in the active Constraint Value field. If, on the other hand, the context is a field for which find data would make little sense (dates, ages, . . .), a message that no find data is available is displayed. If Find is pressed when no context is set, an error dialog box will appear.

- *Clear.* When pressed, the Clear button will cause a query to be reset. All “Fields” check boxes become unchecked, all “Constraint Functions” are set to ‘None’, all “Constraint Values” are set to the empty string, and the contents of the results portion are emptied.

- *Execute.* The Execute button is used to cause the browser to go to the database to fetch the results of the current query.

- *Save.* The Save button is intended to give users the ability to save the results of a query to their local file systems. Because Java, when operating over the Web, does not access the client's file system, results must be sent to the user via an alternate means. A copy of the currently displayed results is emailed to the user in a simple text, tab-delimited format. When the Save button is pressed, a window is displayed to the user prompting for an email address to which to send the results, and a subject heading to be attached to the email message.

- *Quit.* The Quit button serves a dual purpose. It may be used to quit the current query and select a new one, or it may be used to quit the NOIS Browser altogether. When Quit is pressed, the user is presented with a window with two buttons, one that will return the user to the query selection list, and one that will exit the NOIS Browser.

3.3 NOIS Input Forms

The NOIS Input Forms support users in browsing through a series of screens, allowing them to enter new information into the database, or fetch existing data and update them. The forms present the users with fields where data can be entered or modified. The information fetched from the database is displayed in a table; users can scroll through the table and select a row. Menus and buttons are the mechanisms for navigating among the screens and issuing commands to the system. The system guides the users in performing the actions with appropriate cues like displaying help or status or error messages, enabling and disabling menu items, and changing the appearance of the cursor to reflect on-going processing.

The Input Forms are comprised of a set of screens. These screens follow a standard design: every screen can be broken down into a standard set of components. The layout of the screens directly follow the conventions established in the earlier Oracle Forms user interfaces. The reasons for continuing with the same design are its simplicity, flexibility and user-friendliness, and the desire to maintain continuity in the look-and-feel of the interface across versions of the software.

Screen names in NOIS are in the following format: `<Task menu option><type>`. *Task* refers to a database subsystem, and is further described in Section 3.3.2. NOIS screens can be one of two types. The *Type* options are:

- *Select screens.* Select screens are used to select existing records from the database. Information cannot be put in the database from these screens.
- *Entry screens.* Entry screens allow for information to be entered into the database. New information may be added for the first time and existing information may be updated.

Examples of NOIS screen names are "Patient Select", "Patient Entry", "Radiotherapy Entry" and so on.

3.3.1 Screen Layout

Figure 4 shows the different components in a typical select or entry screen. It contains a menu at the top followed by a number of vertically stacked panels.

Starting from the top, the panels in a screen are:

- *Context Panel.* The context panel serves two purposes: on the left corner it displays the name of the screen; on the right it displays the current context for navigation, i.e., values that identify what data is currently being processed. The context is updated every time the user selects records from the database or commits values to the database.

- *Input Panel.* The input panel displays the fields where users enter data. Each field has a name followed by an entry area. The entry area can accept text, dates, or numbers, or can be a checkbox or a pull-down list, where appropriate. It is worth mentioning that select screens also have an input panel, although they do not allow data entry. On select screens, the field values entered by the user are used to constrain the rows that are selected from the database.

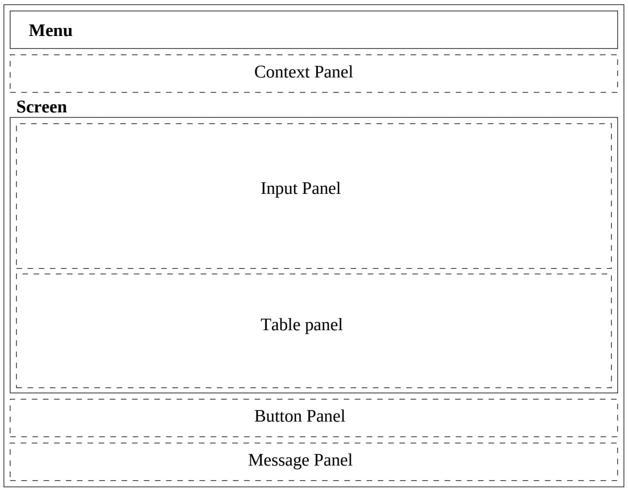

Figure 4: Stacked screen layout

- *Table Panel.* The table panel displays records that are fetched from the database whenever the user presses the Find button. At run-time the table panel contains a scrollable table from which a row can be selected.

- *Button Panel.* The button panel contains a series of buttons that are used to issue commands to the system. The buttons that appear in this panel depend on the screen, and correspond to actions that are appropriate for a screen.

- *Message Panel.* The message panel is used to display help and status messages to the user.

Figure 5 shows the dump of an Input Form screen. The correspondence to the panels in Figure 4 can be observed. This is the "Patient Select" screen from the Radiotherapy forms. The context information shows the values for the fields 'Hospital Patient ID', 'Last Name' and 'First Name'. There are five fields in the input panel for this screen, all of which can be used to limit the number of rows fetched from the database. In this instance, data for 32 patients were fetched from the database; the second row from the table was then selected (it is highlighted in the table panel),which caused the values to be copied in the entry fields.

3.3.2 Navigation Through Screens

The NOIS Input Forms are started in the same way as the NOIS Browser when a Java-enabled web browser is directed to the NOIS input form launcher URL. Each NOIS Input Form is a Java applet that opens windows onto the user's workdesk outside of the web browser window. The first window that appears is a Login/Password window. All users of the NOIS Input Forms must be validated before use.

Upon successful log in, the user is presented with a NOIS welcome screen that contains a Task menu. The tasks correspond to input subsystems, such as patient, surgery, radiotherapy, etc. The user may select a task from among those that are initially enabled in the menu. Each task listed in the Task menu will normally have a select screen and one or more entry screens associated with it. Choosing an option first brings up the corresponding select screen; from there the user can navigate to the entry screen for that subsystem.

The menu items provided to the users are uniform across the input forms. The top level options are:

- *Task.* The Task menu lists different subsystems of the database that can be accessed by the user. In the database, the major subsystems are inventory and clinical, which in turn have smaller constituent parts; it is these smaller parts that appear as options in the Task menu.

  The options under the Task menu depend on the type of Input Form. For instance, in the Radiotherapy forms the Task menu lists 'Patient' and 'Radiotherapy' as the available subsystems. The Task menu options are disabled or enabled to ensure that any incorrect sequence of actions is not performed by the user. In most cases this sequence is dictated by integrity constraints in the database. Continuing

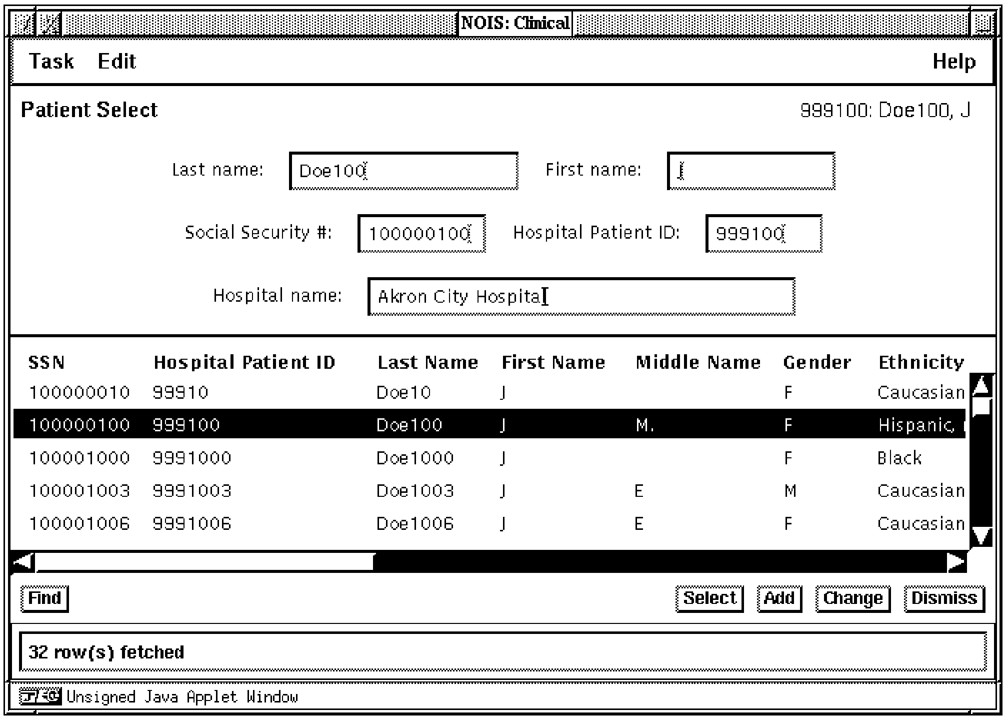

Figure 5: Clinical radiotherapy forms: "Patient Select" screen

with the above example, the option for 'Radiotherapy' remains disabled until a patient is selected; this enforces the requirement that any radiotherapy information must be associated with a patient.

- *Edit.* The Edit features allow users to clear fields in the input panel.
- *Options.* The Options menu list flags that can be set for database operations. Currently, the only available option is to set wildcard matching during constrained selects from the database.
- *Help.* The Help menu provides information about using the Input Forms.

As mentioned earlier, each screen has a button panel. The buttons in this panel are used to issue database commands in the user interface. Buttons are also the means for navigating through the select and entry screens for a subsystem. Switching between subsystems is done from Task menu options. The standard buttons in the Input Forms are:

- *Find.* The Find button appears on both select and entry screens and is used to fetch values from the database into the table panel, in a manner similar to the Find button in the NOIS Browser. On select screens, Find fetches and displays comprehensive information about a subsystem. The amount of information fetched is usually constrained by suitable values. For instance, the Find button on the "Patient Select" screen brings up a list of all patients in the database along with relevant information. This list can be limited only to certain patients based on attributes like last name, social security number, hospital name, etc.

  On entry screens, the use of Find button is related to fields for which values must be selected from lookup tables in the database. When pressed, the Find button brings up a list of valid values for those fields. Thus, depending on the position of the cursor, different lists are displayed in the table panel.
- *Add.* The Add button appears on a select screen and brings up the entry screen for the same subsystem, for purposes of putting new data into the database.
- *Change.* The Change button appears on select screens and also brings up the entry screen for a subsystem, but first a row from the table panel in the select screen must be highlighted. This selects

existing values from the database, which can then be updated. The fields in the entry screens show the values that are being updated.

- *Delete.* The Delete button appears on select screens and deletes records from a subsystem. The record to be deleted has to be selected from the table panel.
- *Ok.* The Ok button appears on the first entry screen of a subsystem, and is used to commit records to the database. It brings up the corresponding select screen.
- *Cancel.* The Cancel button appears along with Ok, and is used to abort a database operation. It also brings up the corresponding select screen.
- *Dismiss.* The Dismiss button appears on select screens. It brings up the screen that the user was viewing before navigating to the current subsystem.
- *More.* The More button is used for navigating forward to the next entry screen for subsystems with multiple entry screens.
- *Back.* The Back button navigates backwards to the previous entry screen for a subsystem.

Apart from these standard buttons, some Input Forms may have additional buttons to provide a sequence of navigation actions that follow the underlying database structure (e.g., Go to Container), or to indicate the high-level semantics of a database operation (e.g., Alias).

4 Object-Oriented Design

We describe next the object-oriented design for the NOIS Input Forms. The primary goal of the design is obviously to achieve the functionality outlined in the previous section. A secondary, but equally important goal is to design components in a way that maximizes their reuse. Reuse not only offers us the advantages of reduced development, testing and maintenance efforts, but also helps us achieve consistency in the behavior of the user interface across forms. The guiding principles for us in pursuit of this goal were that any behavior that is replicated (1) between the Browser and the Input Forms, (2) across Input Forms, or (3) within a single Input Form should be coded only once.

Reusability across the NOIS Browser and the NOIS Input Forms is restricted to login verification, the 'back-end' component, and the Find button. Both the Browser and the Input Forms have to authenticate a user's access to NOIS, and both need to access the Oracle database. The use of the Find button is also same in the Browser and the Input Forms: both use Find to select rows from a database table or view. The objects that implement these functions are shared between the two interfaces, thus meeting our first principle for reuse. Looking at figures 3 and 5 one might argue that the table panel is also shared between the Browser and the Input Forms. The base implementation of the table object is same for both interfaces. However, in the Input Forms, the table object allows selection of rows that triggers a data transfer between objects (explained in Section 5), and this added functionality is not required in the Browser. Hence, to account for this difference, we have created a separate TablePanel class that is derived from the base table class.

Because there is only one NOIS Browser, and not many as is the case for the NOIS Input Forms, and because there is no replication of function within the Browser (principle 3) as it has only one screen, no other reuse issues arise. Thus, in this section we emphasize the design of the Input Forms, giving details of the classes and their relationships. We adopted a top-down approach for the design, starting with a coarse-grained view of the system, identifying classes at that level, and then decomposing them into their constituent parts, thereby successively going to more and more finer-grained views.

4.1 Class Hierarchy

The design process for the forms essentially involved identifying the required classes and their relationships. The decomposition of classes is in terms of domains of responsibility. Each class is made responsible for managing its internal state and for providing certain services. The Oracle version of the Input Forms served as precise specifications of what was required, and this eased the design process considerably. At the top

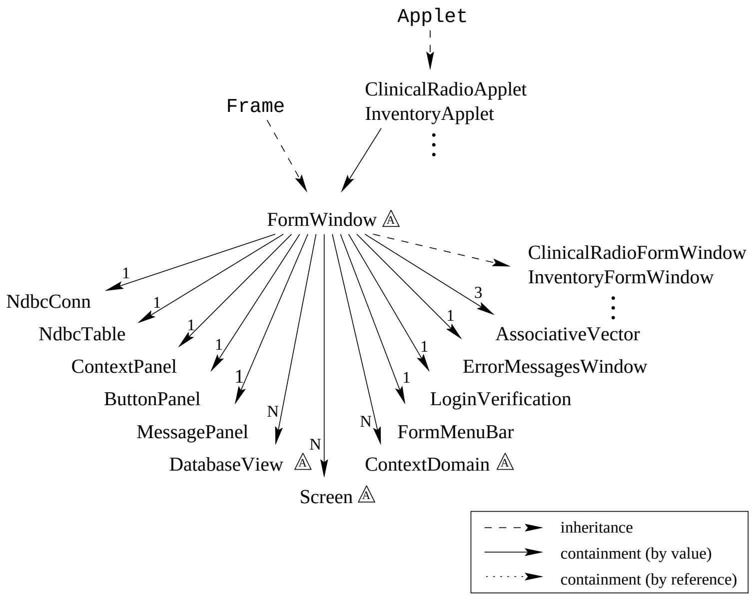

Figure 6: Class hierarchy of Input Forms

level, each Input Form starts off as a Java applet. The Java `Applet` class, thus, forms the root of the class hierarchy. The `Applet` class is subclassed once for each type of Input Form identified in Section 1, in order to launch that form.

4.1.1 The Notation

We will use class diagrams like Figure 6 to illustrate our design. Each class diagram shows the inheritance and containment relationships among classes. A containment relationship is qualified as either containment-by-value, or containment-by-reference, as suggested by Booch [17]. Each of these relationships is shown by a different arrow. The cardinality of a containment-by-value relationship is shown next to the arrow. In most cases, a number shows the exact count of objects that are instantiated; an N indicates a value that is greater than one, but can vary across the subclasses. For example, Figure 6 shows that FormWindow contains N instances of DatabaseView. This means that the concrete subclasses of FormWindow such as ClinicalRadio-FormWindow, InventoryFormWindow, etc., instantiate varying number of DatabaseView objects. Abstract classes in the diagrams are marked by a triangle. NOIS classes and standard Java classes are shown in different fonts in order to make the distinction. Although some of the notation is borrowed from Booch, it is simplified for the purposes of this paper. Details about the internal structure of the classes are also simplified. A more comprehensive documentation is available in HTML at `http://acuity.cis.ohio-state.edu:/doc`.

4.1.2 Form Classes

As Figure 6 illustrates, the `Applet` class is extended as form-specific applets. Each of these form-specific applet class contains a FormWindow. The containment relationship is by-value, which means that a FormWindow object is physically contained within a form-specific applet. FormWindow itself is abstract, so the applets actually instantiate one of its subclasses. FormWindow is derived from the Java `Frame` class and, similar to the applets, is extended for every type of Input Form. It represents the main window that comes up when an Input Form session is initiated.

FormWindow instantiates all other objects that constitute various components of the Input Forms. Among these are: (1) DatabaseView (Section 4.2), representing objects in the NOIS database that are directly accessed by FormWindow, (2) FormMenuBar (Section 4.3), to display a menu, (3) ContextPanel and ContextDomain (Section 4.4), to display information in the context panel of Figure 4, (4) Screen (Section 4.5), corresponding to the panel shown in Figure 4, (5) ButtonPanel (Section 4.6), to display the buttons, (6) MessagePanel and ErrorMessagesWindow (Section 4.7), to show help and error messages, respectively, and (7) the database connection objects: NdbcConn and NdbcTable (Section 6). It should be noted that

FormWindow contains multiple Screen, DatabaseView and ContextDomain objects. The exact number depends on the form type, and is coded in the subclasses of FormWindow. FormWindow defines accessor methods for all its components. It also defines iterator methods for components with containment cardinality N. These methods perform simple actions like clearing all Screens and DatabaseViews.

4.1.3 Design Principles

Before discussing further the individual classes, it is worthwhile to point out a number of features of our design that Figure 6 illustrates, and that will also apply to subsequent class diagrams.

First, concrete classes (or classes that can be instantiated) typically appear as leaves in the inheritance-class hierarchy. The intermediate classes provide a generalization, by grouping together common attributes and services of the classes descending from them. In almost all cases, it is inappropriate to instantiate such classes, and so they are made abstract. In Figure 6, the subclasses of `Applet` and FormWindow form the leaves of the inheritance hierarchy, and are thus concrete.

Second, interaction between classes is handled through references to appropriate classes. If class A needs to access the services of class B, then it does so through a reference to class B. In all such situations, class A is passed a reference to FormWindow as part of its constructor. From this reference, A obtains references to class B, and any other class that it might need to send a message to. Needless to say, reference to FormWindow itself will suffice for all possible inter-class interactions, because FormWindow contains, either directly or transitively, all Input Forms classes. However, we require that each such class obtain a direct reference to the class it needs to communicate with. Though this results in redundant references being kept, the run-time performance improves as the chain of references to be traversed reduces to one. The entire chain is traversed only once, in the constructor, rather than on every access. In the class diagrams, we only show the relevant references, so that it will be clear which classes a given class needs to interact with.

The third observation relates to containment relationships of cardinality N. For example, FormWindow contains N instances of Screen. What FormWindow really contains are instances of subclasses of Screen, and each subclass appears only once. FormWindow treats the subclass instances uniformly by their base class. It stores these objects in an AssociativeVector,[2] where each object is indexed by an identifier. The $identifier \longleftrightarrow object$ mapping enables FormWindow to refer to a specific Screen, given its identifier. To support this identification scheme, each NOIS class defines a static method that returns its identifier. Note that this scheme works because FormWindow does not instantiate any subclass of Screen more than once. Thus, in Figure 6, FormWindow has three AssociativeVectors to store the $identifier \longleftrightarrow object$ mappings for DatabaseViews, Screens and ContextDomains.

With this background, we now look at the specifics of each class instantiated in FormWindow.

4.2 Database Classes

The database-related classes provide a useful abstraction for NOIS database objects like tables and views, that are referenced by the Input Forms. The NOIS database schema follows the relational data model [18, 19]. It contains a number of relations or base tables that hold records of data. The users of NOIS can update data in most of these tables. A few of them, however, serve as lookup tables — a lookup table lists valid values for a field in another table. Lookup tables help us control the values entered for certain fields, through foreign key constraints. Examples of such tables in NOIS are `Hospitals`, `Ethnicities`, etc. The users do not have update privileges on lookup tables.

The NOIS database also defines views of data. The views combine values from several base tables, and exist primarily for query purposes. For each database subsystem, there is one comprehensive view that provides all information relevant to that subsystem. The Input Forms use such subsystem views to update base tables that store data from a subsystem.

We, thus, identify three database objects that need to be modeled by Java classes: (1) the base tables that store data entered through the Input Forms, (2) the tables providing lookup values for a field; we will refer to these as *lookup views* henceforth, and (3) the subsystem views, that we will refer to as *updatable views*. Each of the above three classes is entrusted with certain responsibilities:

[2] An AssociativeVector allows vector indices to be specified as strings. It can, therefore, maintain a mapping between an object and a string.

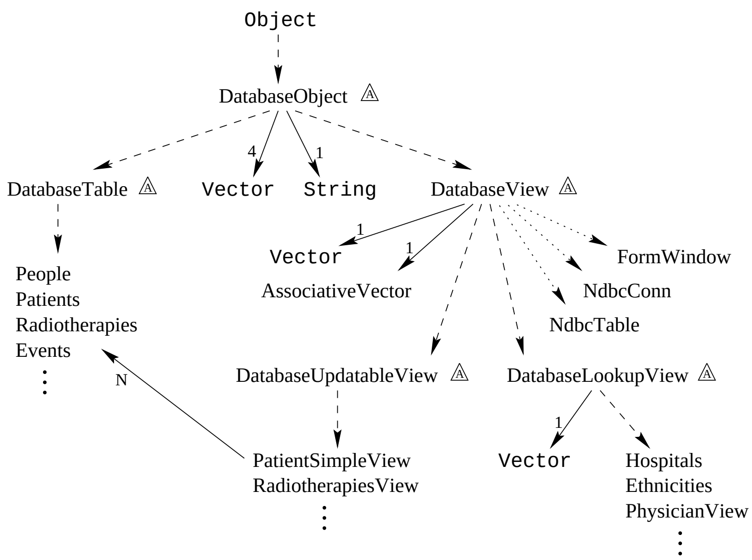

Figure 7: Hierarchy of database-related classes

1. The class for a database table must provide structures to represent a row from the underlying table; this row is either created, updated, or deleted. Additionally, it must transform data from its representation in LabeledComponents (Section 4.5) to a form wherein it can be sent to the database.

2. The class for a lookup view must provide a means of fetching rows from the database, with or without constraints. These rows should be cached to prevent repetitive fetches of the same data. Caching is appropriate for lookup table values because they are typically not very large, and remain static during a user session. The rows retrieved in this class should be accessible to other classes like TablePanel (Section 4.5) and LabeledComponents.

3. The classes for updatable views must also provide services for fetching rows from the database, with or without constraints. However, caching of data values is not appropriate due to the huge magnitude of data from updatable views and its dynamic nature. In addition, each such view is responsible for updating the base tables from which it derives its values. To avoid redundant updates, a view must be able to determine if a user is trying to commit the old values (that were retrieved from the database), without making any changes.

Figure 7 shows the hierarchy of database-related classes. At the root of the hierarchy is DatabaseObject, the base representation for all database entities that we intend to model. Each DatabaseObject has a name to identify it, and is composed of four vectors: three of these together constitute a row from a database table or view — they store field names (*name vector*), field types (*type vector*), and field values (*database vector*); the fourth vector stores the names of primary key fields (*key vector*). There is a one-to-one mapping between the elements of these vectors, and together they represent the current row from a table or view that is being processed.

DatabaseObject is extended as DatabaseTable and DatabaseView. DatabaseTable does not introduce any additional attributes. It provides methods that iterate over its vectors and generate strings for SQL commands like insert, update and delete. DatabaseTable is subclassed for specific tables from the NOIS database, and these subclasses are instantiated at run-time.

DatabaseView extends DatabaseObject to represent views, both lookup and updatable. In addition to the attributes inherited from DatabaseObject, it contains another vector to store values from a row. Thus, a view object has two vectors that store a row. One of the vectors, *panel vector*, stores field values displayed in InputPanel, whereas the other, database vector (inherited from DatabaseObject), stores field values that are fetched from the database. By comparing the field values in its panel and database vectors, a DatabaseView can observe modifications made to the values in a row, and perform only the necessary database commits. Because rows from database views are displayed in the table panel, DatabaseView maintains a mapping between field names and their table panel headers in an AssociativeVector. Apart from these two

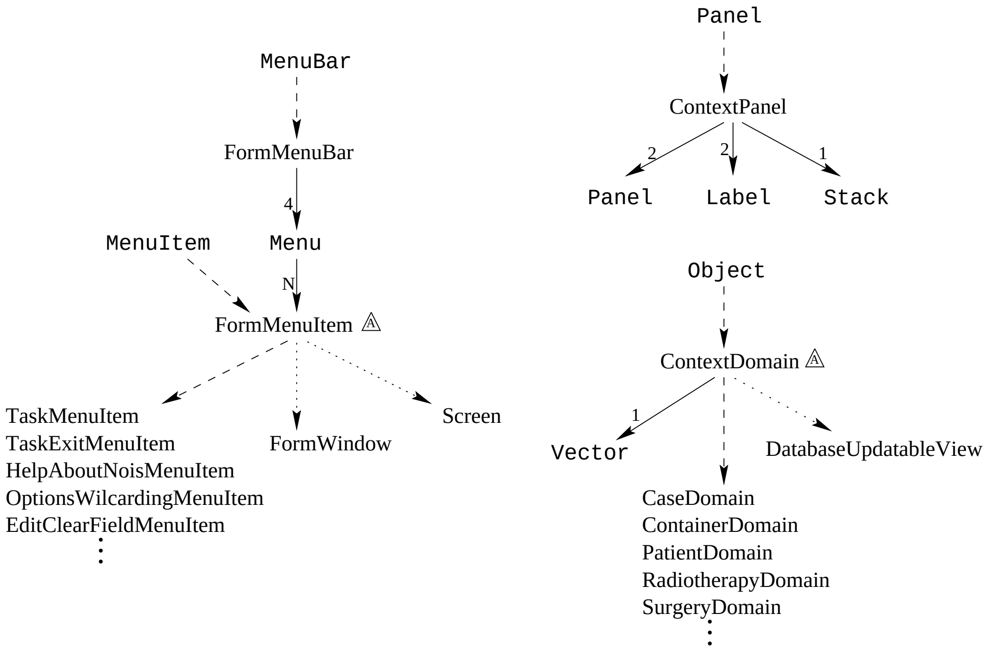

Figure 8: Hierarchies of menu-related and context-related classes

data members that are physically contained within it, a DatabaseView also maintains references to other classes: FormWindow, NdbcConn and NdbcTable. The first reference is required to access MessagePanel and ErrorMessagesWindow when DatabaseView needs to display messages or errors, while the other two are required for performing database operations. Methods in DatabaseView generate strings for the SQL select command.

DatabaseView is further extended to provide specializations for lookup views and updatable views, to account for their functional differences. DatabaseLookupView defines a vector to implement the cache; the dynamic behavior of the cache is discussed in Section 5. The subclasses of DatabaseLookupView are individual lookup tables from the database. Methods in DatabaseLookupView manage its cache.

DatabaseUpdatableView is subclassed for each subsystem view in the database. Each of these subclasses encapsulates the objects for database tables that are updated by that subclass. As was previously mentioned, a subsystem view includes all data pertinent to a database subsystem, and is used to update tables storing such data. A DatabaseTable object thus exists only within a DatabaseUpdatableView. It is possible for a table to belong in more than one subsystem. For instance, the `Events` table in NOIS stores data from both the patient and radiotherapy subsystems. So an Events object is instantiated by both PatientSimpleView and RadiotherapiesView, the views corresponding to patient and radiotherapy subsystems, respectively. Methods in DatabaseUpdatableView update database tables corresponding to table objects contained within it.

4.3 Menu Classes

The menu-related classes, shown in Figure 8 on the left, consist of a FormMenuBar that extends the Java `MenuBar`. FormMenuBar contains four `Menu` objects, corresponding to Task, Edit, Options, and Help menus, that are available in all Input Forms. Each such menu can contain an arbitrary number of `MenuItems`. The standard menu items that appear in all the forms are instantiated as specific subclasses of FormMenuItem. This, in fact, includes all Edit, Options, and Help menu items. Items under the Task menu that appear in only certain Input Forms are instantiated as generic TaskMenuItems by the appropriate FormWindow. All FormMenuItems maintain reference to a Screen that they navigate to on being activated. They also refer to FormWindow in order to access its methods, when activated.

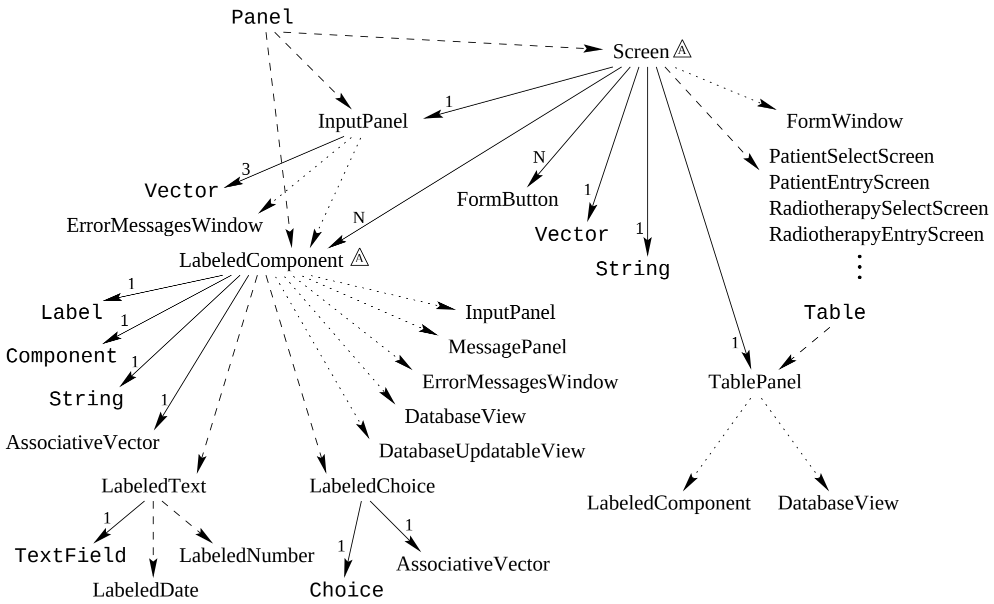

Figure 9: Hierarchy of screen-related classes

4.4 Context Classes

The context-related classes are responsible for displaying information in the context panel. As the right side of Figure 8 shows, there are two main classes that accomplish this. A ContextDomain maps to a database subsystem, and extracts information from that subsystem that needs to appear in the context panel. It maintains a reference to a DatabaseUpdatableView and a list of fields from that view that constitute the context information. ContextDomain is subclassed once for each subsystem in the database.

ContextPanel extends the Java `Panel` class, and is packed within a FormWindow at run-time according to the layout in Figure 4. It contains two `Panels` that display `Labels` for screen name and the current context information, and provides methods for updating them.

The context information is manipulated through operations on a *context stack* — a stack of ContextDomains. The string for current context information is built by traversing the context stack bottom-up, and concatenating information from each domain that appears on the stack. The pattern of users' navigation in the Input Forms follows a stack-like manipulation of ContextDomains; hence the use of a stack. The context stack starts out empty, and as users select information from the database, domains are pushed onto or popped off the stack. For example, when users select a patient, PatientDomain is pushed onto the context stack, and relevant information from that domain is displayed in the context panel. This situation is depicted in Figure 5. Once the user selects a radiotherapy for the current patient, RadiotherapyDomain gets pushed onto the stack, and the the context panel shows information from both domains. If now the user selects a surgery for the same patient, then RadiotherapyDomain is popped from the context stack and SurgeryDomain is pushed onto it.

4.5 Screen Classes

The screen-related classes comprise of input and table panels shown in Figure 4. They are responsible for (1) collecting data from the user through fields in the input panel, (2) displaying information fetched from the database in the table panel, and (3) performing certain constraint checks on the values entered by the user to validate their integrity.

Screen appears at the top of the hierarchy in Figure 9. Its components include an InputPanel and a TablePanel. It instantiates a series of LabeledComponents and FormButtons that appear in the input panel and button panel, respectively, in the user interface. It keeps the references to FormButtons in a vector. The references to LabeledComponents are kept by InputPanel. The lists of LabeledComponents and FormButtons are screen-specific, and are constructed in the subclasses of Screen. A Screen's name is stored as a Java `String`. Each Screen maintains a reference to FormWindow in order to access its components.

An InputPanel exists within a Screen. Once instantiated, it receives references to all LabeledComponents from its containing Screen, and stores them in a vector. It also keeps track of the *active* LabeledComponent — one that has the input focus. InputPanel contains two other vectors to store information regarding database constraints associated with its LabeledComponents. The two kinds of constraints are: (1) *required* fields — fields that must have a value, and (2) *related* fields — fields A and B are related if either both must have a value, or both must be null. All this information is passed to InputPanel by Screen after it is instantiated by Screen. The methods in InputPanel iterate through its list of LabeledComponents, refreshing their values, and performing format and database constraint checks. If any of the validation checks fail, InputPanel displays an error message using its reference to ErrorMessagesWindow. InputPanel thus discharges responsibilities 1 and 3 entrusted to Screen.

A LabeledComponent represents a field in the Input Forms. As the name suggests, it comprises of a `Label` to display a field's on-screen name, and a `Component` that forms the text entry area. A `String` stores an optional message that is displayed in the message panel when a LabeledComponent becomes active. Each LabeledComponent receives its value from, and sends it value to, a field from a database view. It thus maintains a *source view reference* to a DatabaseUpdatableView, and a *source field reference* that a points to a field within that view. (Recall that a DatabaseUpdatableView has two vectors that store field values: database vector and panel vector. A LabeledComponent interacts only with the panel vector through its source field reference.) As an example, the LabeledComponent for 'Last Name' from the "Patient Select" screen, shown in Figure 5, points to a PatientSimpleView object as its source view reference, and stores a corresponding field name from that view as a `String` object. In addition to the above references, a LabeledComponent maintains a *find view reference*. This reference is required for those LabeledComponents that can potentially be used to constrain rows selected from the database because they (1) appear on select screens, or (2) have a lookup view associated with them in the database. For the first case, the find view reference points to a DatabaseUpdatableView, and is the same as the source view reference. In the above example, the find view reference for 'Last Name' also points to a PatientSimpleView object. For the second case, the find view reference points to a DatabaseLookupView. An example of this is the LabeledComponent for 'Physician Name' field on the "Radiotherapy Entry" screen. While its source view reference points to RadiotherapiesView, its find view reference points to PhysicianView. For the case where a LabeledComponent's source and find view references are different, the LabeledComponent also keeps a *find field reference* that is similar to its source field reference: it points to a field within a DatabaseLookupView. In case of LabeledComponents that do not fall in either of the above two categories, the find view reference is set to null.

Table 1 summarizes this discussion by providing a classification of LabeledComponent types. 'Last Name' when it appears on "Patient Entry" screen, is an example to Type 1 LabeledComponent, and is used to input values to the database; on entry screens, 'Last Name' becomes an instance to Type 2 LabeledComponent and is used to fetch data for selective patients (those with a certain last name) from the database. Notice that a LabeledComponent of Type 3 can appear on either select or entry screens. In both cases, however, its find view reference points to a DatabaseLookupView, so that when Find is pressed with a Type 3 LabeledComponent being active, rows from its find view reference, suitably constrained, are fetched and displayed in the table panel. On select screens, the highlighted value from these rows is also used to constrain values from the source view reference. For example, consider the behavior of 'Hospital Name' that appears on "Patient Select" and "Patient Entry" screens in the Radiotherapy forms, and in both instances its find view reference points to Hospitals. When Find is pressed with 'Hospital Name' as the active LabeledComponent, the list of hospitals from the database is brought up in the table panel; the user can select a hospital from this list. This behavior is common on both select and entry screens. On select screens the selected hospital is then used to constrain patients selected from PatientSimpleView (the source view reference for 'Hospital Name') when a Find operation is initiated with input focus on a LabeledComponent other than 'Hospital Name'.

Each LabeledComponent has references to MessagePanel and ErrorMessagesWindow. The methods in LabeledComponents perform data transfers to and from DatabaseViews that are pointed to by their source and find view references.

LabeledComponent is extended as LabeledText and LabeledChoice. LabeledText uses Java's `TextField` to provide free-form text entry, and is further specialized to accept only dates and numbers. Both these extensions provide methods to ensure that an entered value conforms to proper formats. InputPanel uses these

Table 1: Classification of labeled component types.

Labeled component type	Screen type	Constrains database fetches	Source view reference	Find view reference
Type 1	entry screens	No	DatabaseUpdatableView	null
Type 2	select screens	Yes (from updatable view)	DatabaseUpdatableView	DatabaseUpdatableView (same as source view reference)
Type 3	entry screens	Yes (from lookup view)	DatabaseUpdatableView	DatabaseLookupView
	select screens	Yes (from lookup and updatable views)		

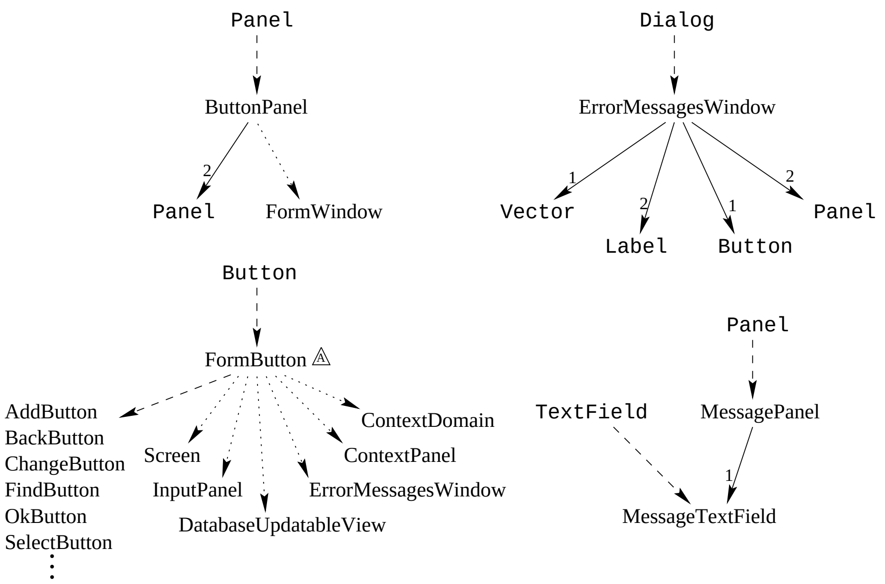

Figure 10: Hierarchies of button-related and message-related classes

methods in performing its functions. Note that LabeledText is an exception to our first design guideline from Section 4.1.3 in that it is instantiated despite being an intermediate node in the inheritance hierarchy. The reason it appears as an intermediate node is that LabeledNumber and LabeledDate are really specializations of LabeledText — the text that can be entered in them is more restricted and needs to validated for proper format.

LabeledChoice uses Java `Choice` to bring up a pull-down list of choices. For each option in the list, LabeledChoice stores a mapping between the option's on-screen name, and the name that gets stored in the database. For example, the LabeledChoice for 'Smoker' on "Patient Entry" lists 'Yes', 'No' and 'Unknown' as the available options. These values are actually stored as 'Y', 'N', and null, respectively, in the database.

TablePanel extends the `sunw.admin.avm.base.Table` class provided with Java Management API,[3] and brings up a list of rows from the database when the Find button is pressed. TablePanel maintains a reference to a LabeledComponent — one that has the input focus when TablePanel tries to refresh itself. TablePanel accesses the find view reference of this LabeledComponent in order to fetch rows from the correct lookup or updatable view in the database. The TablePanel, thus, performs the second responsibility of Screen.

4.6 Button Classes

The button-related classes, shown in Figure 10 on the left, include ButtonPanel and FormButton. ButtonPanel contains two Java `Panels` within which it packs the buttons. The two panels are stacked horizontally so that buttons appear on the left and right sides of the screen (for example, see Figure 5).

[3]Documentation of this class is available at
`http://www.javasoft.com/products/JavaManagement/documents/java_docs/Package-sunw.admin.avm.base.html`.

FormButton extends the Java `Button` class, and provides a means by which users issue commands to the system. Each FormButton contains a reference to its containing Screen, and the screen to which it navigates. In cases where a button is not designed to navigate to a different screen, the latter reference trivially points to the containing Screen. FormButton also contains references to an InputPanel, a DatabaseUpdatableView, a ContextPanel, a ContextDomain, and an ErrorMessagesWindow. FormButton uses these references to update the database, update context, and display error messages. Each type of standard Input Form button identified in Section 3.3.2 is instantiated as a subclass of FormButton.

4.7 Message Classes

The message-related classes provide a mechanism for the system to provide feedback to the users. MessagePanel is used to display help and status messages. It does this by declaring and instantiating an inner class, MessageTextField, and defining methods to control its contents.

ErrorMessagesWindow is the other major component in this group of classes. It displays error messages through a modal dialog box. It contains two `Panels`, packed vertically within a dialog box; each `Panel` contains a `Label` that displays a line of text. A vector within the class holds error messages corresponding to different kinds of errors. ErrorMessagesWindow also instantiates a `Button` to accept user acknowledgment of an error.

4.8 Component Reuse

Having described the static class hierarchies of the NOIS Input Forms in detail, we now briefly revisit our design goal of achieving code or component reuse. The preceeding descriptions demonstrate code reuse by inheritance, and code reuse by instantiation. As an example of code reuse by inheritance, DatabaseLookupView inherits data members and methods from DatabaseView (and transitively from DatabaseObject), and defines only those data members and methods that are unique to itself.

Code reuse by instantiation is realized when multiple instances of the same class are created. This type of reuse is not limited to within a specific Input Form, but also extends across Input Forms. Most of the Input Forms classes support both intra-form and inter-form reuse by instantiation. For example, ContextPanel and MessagePanel are used in every screen of each Input Form. Some of the classes support inter-form reuse only. Examples of such classes are PatientSimpleView and PatientEntryScreen that are instantiated in the Clinical_Patient and the Radiotherapy forms. Finally, there are classes that are form-specific and therefore, do not support inter-form reuse: these are specific buttons (for example, Go to Container in the Inventory form) and menu items (for example, Chunks in Inventory Form menu) that appear in only certain Input Forms. Such classes are typically instantiated only once within an Input Form, and thus have no scope for reuse by instantiation. The three types of classes, namely those that support (1) inter-form and intra-form reuse, (2) inter-form reuse, and (3) form-specific classes, form a spectrum of reuse by instantiation. The classes belonging to the first category offer maximum potential for reuse, whereas those belonging to the third class offer the minimum, often no potential for reuse; the second category of classes offer an intermediate potential for reuse. In our design a majority of the classes fall into the first two categories, and we thus achieve substantial code reuse.

5 Dynamic Behavior

The dynamic behavior of the Input Forms captures all aspects of its interaction with users at runtime. This includes listening for user commands, performing the requisite actions corresponding to different commands, and providing feedback to the users about status and results of the commands.

5.1 Common Patterns of Interaction

The dynamic behavior of the Input Forms is driven by user–generated *events*. An event is generated when a user performs an action in a screen, such as selecting a menu item, pressing a button, or entering text in an entry field. Java's event handling mechanism registers such events and dispatches them to the designated *listener* for that particular action. Any Java class can register itself as a listener by declaring the events about which it wishes to be notified as part of its signature. A listener specifies a set of methods that get invoked in response to the particular events for which it is listening.

Among the Input Form classes, FormButtons, LabeledComponents, TablePanel, and FormMenuItems designate listeners. It is precisely by specifying the behavior of these listeners that we are able to identify and encode patterns of interactions that are common among objects in the Input Forms. Thus, it is in through this activity that we transform our design and implementation approach from one that emphasizes reuse of static objects into one that includes reuse of patterns of interaction among objects.

For example, consider an abstract specification, i.e., not tied to one screen instance, of the actions of the listener for an Add button.

```
1. if not all fields empty, then error message (MESSAGE ID)
2. clear context if one exists ()
3. screen update (screen id)
4. clear panel vector (panel vector id)
5. refresh labeled components (input panel id)
```

When we implement this behavior in Java, we will have encoded explicitly the pattern of interaction of the Add button object with the ErrorMessagesWindow (the method call in action number 1) , the ContextPanel (no. 2), the Screen (no. 3), the DatabaseObject (no. 4) and the InputPanel (no. 5) objects.

This behavior changes from screen-to-screen not in the pattern of interactions of the Add button with other Input Form objects, but only in the set of parameters that are specified in the method call when the Add button is instantiated on a particular screen. For example, if an Add button is instantiated on a "Patient Select" screen in the Radiotherapy Input Form, then for action number 3, the screen id parameter is set to the id of the Patient Entry screen in the Form.

We have identified the following, complete set of listeners in the Input Forms:

- Buttons: when depressed
  - Find
  - Select
  - Add
  - Change
  - More
  - Ok
  - Cancel
  - Back
- Labeled Text Components: when focus gained or lost
- Labeled Choice Components: when focus gained or lost
- Table Panel: when row selected
- Menu Items: when selected
  - Exit
  - Clear Field
  - Clear All Fields
  - Enable/Disable Wildcarding
  - About NOIS
  - Close
  - (other menu items, depending upon Screen/Input Form)

For each listener, we specified its behavior in a manner similar to that shown for the Add button above. In three cases, i.e., for depressing the Find button, for focus lost in a labeled text component, and for row selected in a Table Panel, we had to create subclasses so that we could accommodate a specification of a behavior that was not completely uniform across all screens, but that changed depending on what type of labeled component was last active before the listener was invoked.

These specifications of common patterns of interaction among the Input Form objects were programmed in Java code. They explicitly implement these patterns of interaction and are available for reuse in every instance of an Input Form.

Rather than going into the details of all actions performed by the listeners, we next provide a high level view of how data flows in the Input Forms, from the database to the user interface (reverse dataflow) and from the user interface to the database (forward dataflow).

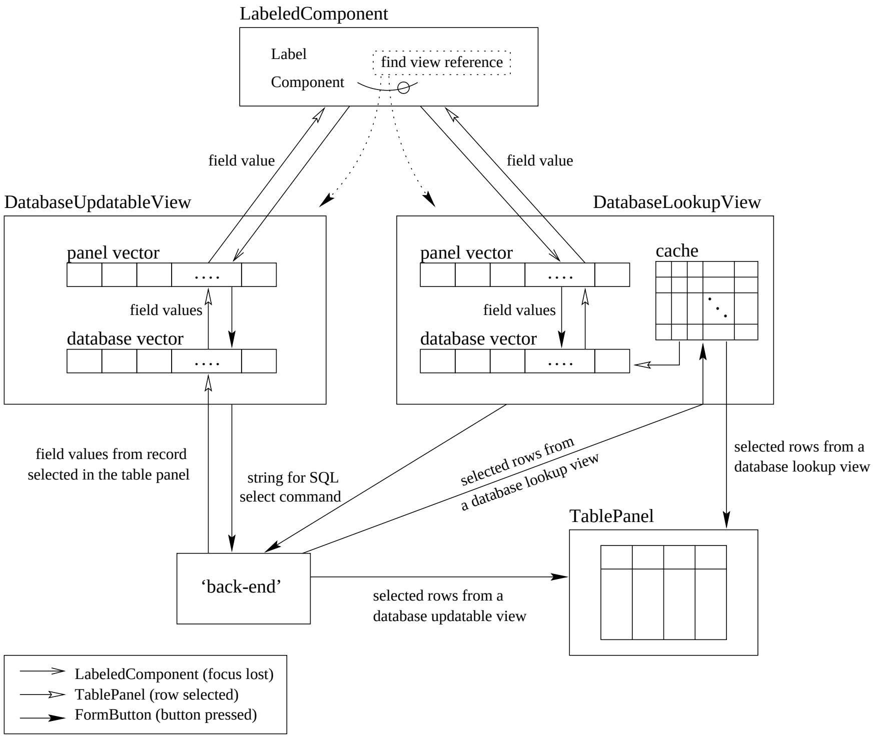

Figure 11: Data flow from the database to the user interface.

5.2 Reverse Dataflow

Reverse dataflow originates at the database and terminates at the user interface. It corresponds to the execution of a SQL select command in the database. The selected rows, retrieved from an appropriate lookup or updatable view in the database, are first displayed in the table panel. Then the row highlighted in the table panel is copied to the labeled components.

Figure 11 shows reverse dataflow through the components of the Input Forms. The legend lists the listeners (and their events) that cause dataflow along various edges in the figure. The dataflow proceeds in three phases that are triggered by different events. Initially, on losing focus, each LabeledComponent transfers its value to the panel vector of its find view reference. This constitutes the first phase of reverse dataflow. The second phase is initiated when the user presses the Find button. The Find button accesses the correct DatabaseView object through its reference to InputPanel, by looking up the active labeled component. The active LabeledComponent can be of Types 2 or 3 for a valid Find button operation. The find view reference of this LabeledComponent maps to a lookup or updatable view in the database. Figure 11 shows dataflow for both cases: whether the *find view* is a DatabaseUpdatableView or a DatabaseLookupView.

Within the find view, the panel vector values are copied to the database vector and a SQL select command is generated, with appropriate constraints based on non-null field values in the database vector. This command is sent to the 'back-end'. The 'back-end' responds with the rows selected from the database. These rows are next displayed in the table panel, if the find view is an instance of DatabaseUpdatableView. If, however, the find view is an instance of DatabaseLookupView, then the fetched rows are first stored in the DatabaseLookupView's cache, and then displayed in the table panel. The final phase of reverse dataflow starts when the user selects a row from those displayed in the table panel. This causes the database vector of the appropriate find view object to be refreshed by field values of the highlighted row. The selected row is either transferred from the cache (in case of a DatabaseLookupView), or from the the 'back-end' (in case of a DatabaseUpdatableView). Notice that the selected row cannot be transferred from the table panel because table panel does not display every field from a database view. From database vector, the field values are copied to panel vector, and then to the LabeledComponents. The data transfers in the third phase are effected by TablePanel.

A DatabaseLookupView's cache is initially empty. In cases where a Find operation is initiated for a

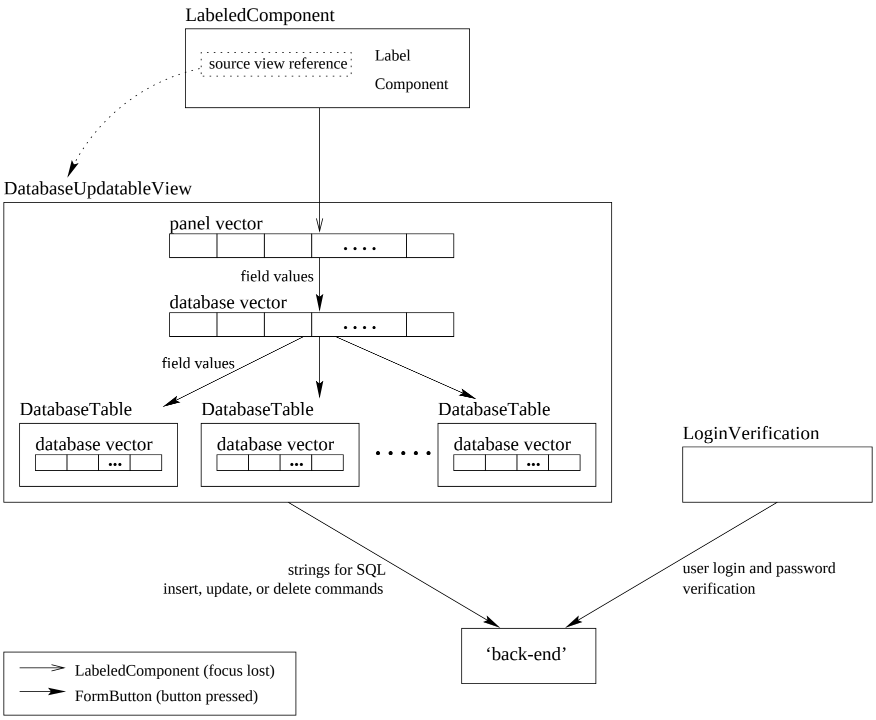

Figure 12: Data flow from the user interface to the database.

lookup view, only the first such operation on a lookup view actually goes to the 'back-end'. This operation populates the corresponding DatabaseLookupView's cache, and all subsequent requests are then met within the 'front-end' (from the cache). Even if the first Find operation is constrained, *all* rows are fetched from the lookup view and stored in the cache; the rows selected by the constraints are returned from the cache.

5.3 Forward Dataflow

Forward dataflow causes relevant data to flow from user interface to the database, and translates to the execution of an insert, update or delete command in the database.

Similar to reverse dataflow, forward dataflow also starts with a 'focus lost' event, which causes each LabeledComponent to transfer its value to the panel vector of its source view reference. The next phase of the dataflow in initiated by a 'button pressed' event, when either an Ok or Delete button is pressed by the user. The appropriate DatabaseUpdatableView object copies its panel vector values to its database vector. From this vector, it partitions the values among DatabaseTable objects encapsulated within it, and invokes a method on each DatabaseTable to get back the string for a SQL insert, update, or delete command. Finally, it passes these strings to the 'back-end'. The 'back-end' executes the commands in the database.

6 Database Connectivity

The previous sections provide a complete discussion of the 'front-end' as shown in Figure 1 in terms of both design and implementation. This brings us to the 'back-end': the subject of this section. The 'back-end', or the NDBC (NOIS Database Connectivity), provides the required link between the user interface on the client machines, and the centralized database on the server machine. The core functionality of the 'back-end' is to enable users to update information in the NOIS database. It receives commands and parameters from the 'front-end', posts them to the database server, and makes interpreted results available to the 'front-end'.

6.1 Back-End Design Options

The architecture of the 'back-end' consists of three distinct building blocks. The first is the set of classes, implemented in Java, that execute on the client machines. The second is the set of programs and data structures that comprise the communication between the Java client-side code and the SQL database server. The third is the SQL database server itself.

Of these three building blocks, the client-side classes follow the object-oriented model of the rest of the Input Forms and so their Java-based implementation is a natural extension of the architecture for the general Input Forms design. The choice of the Oracle Relational Database Management Server product as SQL database server was determined long before this current, latest implementation of the Input Forms and NOIS Browser, so its implementation was pre-determined as well. Only the design of the mechanism for communication between the Java back-end classes and the database was an open issue when we moved to the new Forms/Browser implementation.

Our primary criteria for deciding on the communication mechanism were (1) the mechanism had to allow our clients who were behind firewalls to communicate with the database server, and (2) the mechanism should provide developers with flexibility and control to tailor the connection to our specific needs so that we could start simply and extend functionality as needed.

At the time we had to decide on the implementation of the communication mechanism, there were essentially four options possible. The first was to use the Oracle WebServer, in its full functionality, to provide the connection. The version of the Oracle Web Server available at the time of our design decision required that all requests to the database be in the form of stored procedures or functions in the database. Although we now have a set of stored procedures defined on the database, at the time we had none. Even now, not all of our accesses are in that form. So, that particular requirement was too restrictive for our purposes and precluded this choice. We do use a small part of the functionality of the Oracle Web Server, as a means for a client to establish a URL connection to the server.

A second option for a communication mechanism was to adopt one of several commercial or free-ware Java Database Connection (JDBC) packages. The implementation of these connection packages at the time was socket-based (although this appears to have changed more recently). In general, socket-based connections are not able to penetrate firewalls: a client behind a firewall is not permitted to open a socket on a host outside the firewall. So, this option was not a viable one for us.

Similarly, a third option of implementing from scratch our own socket-based connection had to be eliminated because of firewall considerations.

A fourth option, the one we chose, was to implement the connection mechanism as a series of server processes: an Oracle WebServer to establish a firewall-penetrable URL connection from the client side, paired with a Perl CGI [16] program to handle decoding the request and forwarding it on to the database server.

Figure 13 shows the structure of the 'back-end' and the flow of information among the different elements. The Java components of the 'back-end' are grouped under `nois.ndbc` package, and comprise of two classes: NdbcConn and NdbcTable. These classes define the interface to the 'back-end', and execute on the client machines. The server processes involved in the information flow are Oracle WebServer, the CGI program, and finally the SQL database server — the final destination of all requests to the 'back-end'.

6.2 Details of the NOIS Back-End Architecture

The two Java classes define interface methods to access the services of the 'back-end'. NdbcConn is the primary connection object that is required for all database accesses. Once it has been instantiated, a call to its `open()` method validates user login and password (for the database), and prepares the groundwork for subsequent database operations; this involves initializing its internal data structures. Other methods define the database operations available to users. For instance, `insert()`, `update()`, and `delete()` perform the named operations on database tables; `executeFunction()` and `executeProcedure()` access stored SQL functions and procedures in the database server. Because each of the above command can be executed in batch and non-batch modes, there are methods to set, commit, and abort batches. There are also methods to change user password, and fetch sequence numbers from the database. A sequence is a database object that generates unique, monotonically increasing integers [4]. The NOIS database uses such sequences to generate primary key values for relations. The internal structure of NdbcConn is characterized by a number

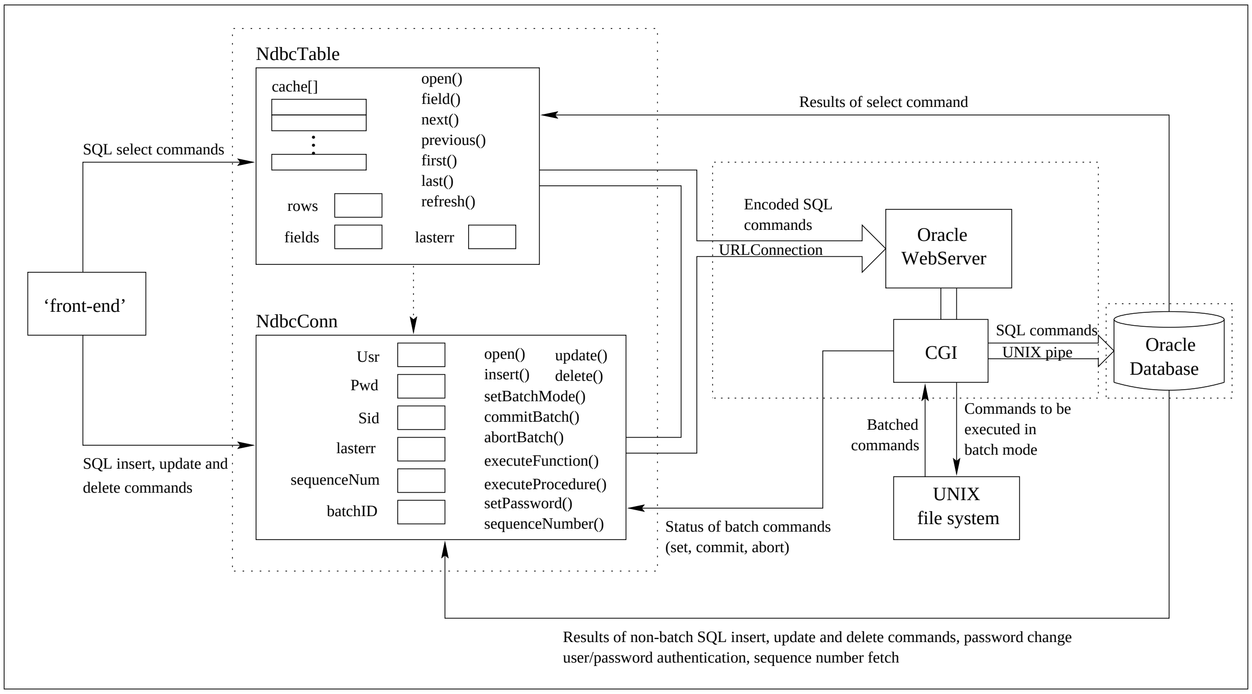

Figure 13: Back-end Architecture.

of member variables that store values like user login and password, rows affected by a database operation, sequence number fetched from database, identifier for a batch mode, and error message in case of errors.

NdbcTable provides for select capability on the database. It accepts select commands from the 'front-end', and sends them to the server. When the results are available, it extracts the selected rows and stores them in internal data structures. The front-end can access these values through method calls. NdbcTable contains a reference to an NdbcConn object (to access user login and password), and other attributes to indicate the number of rows fetched from the database, the number of columns or fields in each row, and error messages for unsuccessful operations.

The Java classes establish connection with Oracle WebServer using a `URLConnection` object that is included in the `java.network` package. They pass all relevant information in an encoded form, as part of the URL, to the server. The encoding corresponds to the GET method in the CGI specification. Each URL connection carries the following parameters to the server: (1) the SQL command, (2) user login and password, (3) the database to connect to, (4) the batch identifier, and (5) a string to identify the requested operation. The Web server invokes the CGI program in response to a connection. The CGI program then decodes the parameters and performs the requested operation.

The 'back-end' functions in two modes, the non-batch mode and the batch mode. In the non-batch mode, SQL commands sent to the 'back-end' are executed immediately in the database, whereas in the batch mode, the commands are stored for later execution. Each set of batched commands constitutes a single atomic action in the database. The need for the batch mode is driven by the structure of our database rather than efficiency concerns. A singe request from the user interface can potentially affect multiple tables in the database, and in order to preserve data consistency, these multiple updates have to be committed in a single atomic step.

For a non-batch-mode command, the CGI program opens a UNIX pipe to the SQL server and writes the command to the pipe. The Java classes can read the output from the database through the URL connection object. They parse the response to ensure that the command executed successfully. When the CGI program encounters a command to set the batch mode, it opens a time-stamped file on the server file system. The time-stamp serves as the batch identifier. All subsequent update requests that arrive with a batch identifier are written to the corresponding batch file. Non-batch-mode commands contain a pre-determined value for the batch identifier field to distinguish them from batched SQL commands. The CGI program continues to augment a batch file till it is told to commit the batch. At this point, it reads the batch file and posts the commands to the SQL server, again via a pipe. It also instructs the server to restore the database to its previous state, if any of the commands fail, to enforce the atomic nature of the batched commands.

One significant aspect of the design is that the database connection is not session-oriented, i.e., it does not remain established through the duration of a user session. Rather, each database access within a session results in a new URL connection to the server. Persistence of data across sessions, when necessary, is achieved by means of batch files on the server. This follows from our decision to go with a CGI-based solution. Other connection mechanisms. e.g., the Oracle Web Server or a JDBC package, may have solved this problem, had we had the option to use them.

7 Experience Using the NOIS Framework

We developed and debugged the framework by creating the Radiotherapy Input Form as an application. After the framework was complete, we used it to develop all additional forms required for the project, excluding research Input Forms.

- Clinical_Patient: having Select/Entry screens for each of Patient, Chemotherapy, Radiotherapy, Surgery, Autopsy, and Non-Treatment Interaction.
- Clinical_Tumor: having Select/Entry screens for each of Tumor, Surgery, Related Treatments, and Last Followup.
- Inventory: having Select/Entry screens for each of Case, Alias, Container, Chunk, Coordinate, Stock and Piece (the latter being increasingly smaller portions of tissue as it is dissected and having specific importance to brain-tumor pathologists).
- Set_Case: having Select/Entry screens for each of Case and Diagnosis.

The original framework developers played the role of framework users while using the framework to implement these four forms. Part of the effort required to generate the four forms was actually devoted to enhancing the framework itself. They discovered two instances where the framework was not generalized properly. In one, what should have been specified as variable parameters had been coded as constants, and in the other a further specification of a class into subclasses was required so that behavioral differences among listeners in that class could be expressed more cleanly and easily. They also discovered a need for one new listener. Such periodic maintenance of frameworks is to be expected because of their generic nature; frameworks can only be tested and debugged in the context of specific applications [14]. Therefore, a new application might suggest a modification or an adaptation of the framework that was not apparent before. Such periodic maintenance of frameworks has been cited as one of its weakness; such modifications, however, have been minimal for us.

Another commonly cited problem with frameworks is the lack of suitable documentation for new users of the framework [15]. While the documentation of the framework code is available via this document and online, we have also documented the process of using the framework to generate a new Input Form.[4] We had an opportunity to ask students who had not been involved in the development of the framework to use the framework to instantiate a new form, strictly as an exercise. This new form, the Chemotherapy Input Form, is comprised of Select/Entry screens for Patient and Chemotherapy.

7.1 Lack of Metrics for Framework Design

Based on this experience, the NOIS development team unanimously agreed that the use of a framework approach while developing the latest version of our Input Forms dramatically improved our reuse of code and, of even greater importance to us, dramatically improved the quality of the code as judged by the number of errors found while debugging.

We surveyed the literature for appropriate object-oriented metrics that would allow us to quantify our experience and to relate it to others in a manner that might be as convincing to them as it was evident to us.

[4] See `http://acuity.cis.ohio-state.edu:8888/using_framework.html`

7.1.1 Metrics for Reuse

For reuse, we found several suggestions for possible measures[20]:

- Verbatim reuse (absolute): the number of times a class is reused as is.
- Leveraged reuse (absolute): the number of times a class is reused with adaptation.
- Verbatim reuse (relative): the percentage of reused, as is, classes.
- Leveraged reuse (relative): the percentage of reused, with adaptation, classes.

One important property of a metric, *noncoarseness*, is that not every class can have the same value for a metric, otherwise the metric has lost its value as a measurement [21]. The metrics defined above are not noncoarse with respect to our framework. Each class in the framework is reused, as is, in every application.

When generating a new Input Form, the primary task of the developer is to enumerate and instantiate all the relevant screens and the objects specific to each screen and to specify the particular parameters that change in each listener on each screen. For example, for an Add button on a "Surgery Select" screen in the Clinical_Patient form, the parameter to the method that updates the screen has to be set to the "Surgery Entry" screen, and so on.

Except for certain extensions to the framework itself as mentioned above, the developer reuses all existing classes, as is. No new methods are written and none is modified. This type of reuse is precisely the goal of the framework method, and our particular project was such that we were able to achieve this goal 100% of the time.

7.1.2 Metrics for Framework Design

Just as we found existing reuse metrics to be of no value for our framework, we also found the metrics proposed for object-oriented design to be of little value, particularly when they are used as quality indicators.

Some of the metrics proposed for evaluating object-oriented design include [21]:

- Weighted methods per class.
- Depth of inheritance tree.
- Number of children.
- Coupling between object classes (typically interpreted as an object in one class calling methods defined in an object in another class).
- Response for a class (set of methods that can potentially be executed in response to a message received by an object of this class).
- Lack of cohesion in methods (the extent to which different methods in a class operate on the same set of instance variables).

Basili, Briand and Melo investigated these metrics as quality indicators by generating a set of hypotheses relating these metrics with fault-proneness [22]. In general, they hypothesize that the simpler the design, the less fault-prone the resulting application is, and so the higher the quality. For example, they conclude that classes with a smaller number of methods are of better quality than those with a larger number, classes with less children are higher quality than those with many, and so on.

They also conclude that "Highly coupled classes are more fault-prone than weakly coupled classes because they depend more heavily on methods and objects defined in other classes." This is of particular interest to framework designers because a primary design goal in a framework is to design and code highly coupled classes, i.e., to explicitly encode in the framework the patterns of interactions among the classes.

These metrics appear to be very useful if the goal is to evaluate an object-oriented design that leads to the implementation of a single application. However, they appear to be less useful to evaluate designs that lead to the implementation of an object-oriented library targeted for reuse.

When viewed as quality indicators, these metrics tend to emphasize simplicity in the design of the components themselves, those that would populate a library for reuse, at the expense of the design and implementation of applications that later may be generated from the components.

If in fact the patterns of interaction among the components is complex, because the application itself is complex, then handling the complexity is delayed until an application is generated using the component library. This delay works to destroy the fundamental purpose of the component library, i.e., to provide high-quality components that shield the application programmer from the inherent complexities in the problem domain.

Thus, we find that these metrics are not especially useful for evaluating framework design because a framework should be evaluated not with respect to itself only, but with respect to itself *and* the applications that are generated from it. New metrics are needed that combine attributes not only of the class structure targeted for reuse, but also of those applications that are generated from the reuse components.

7.2 Experience Using the Framework: Narrative Account

Faced with a lack of metrics to evaluate our framework design, we present a narrative account of our experience. The account relies are hard-to-measure attributes like programmer time and skill. However, we believe the overall tendency of the narrative supports our generalizations about successful framework design and use.

Each member of the development team is assumed to work approximately 20-hour weeks. Sometimes one person is solely responsible for a single Input Form, but sometimes 2 or even 3 people are assigned. Especially when we moved to object-oriented development, significant time was spent in team-design efforts. The person-weeks that we quote are loose estimates of one conglomerate 40-hour person-week.

Our implementation and debugging time for each Oracle-based input forms ranged from 6-10 person-weeks, depending upon the complexity of the form and the skill and experience of the programmer.

We then moved to an object-oriented approach that emphasized static object definitions and a static class hierarchy. In this effort we devoted approximately two person-weeks to the design process and we reduced the development time for each form by several person weeks. However, the debugging effort was similar to that for the Oracle-based forms, so that we didn't see a dramatic reduction in the total cost of development and we saw little increase in the quality of the input forms.

It was not until we moved to the framework concept, combining reuse of static objects and reuse of patterns of interaction among the objects that we experienced a dramatic, overall improvement, particularly in the quality of the Input Forms.

Our estimate of the time it took to develop our first version of the framework is 28 person-weeks. The development included:

- A refinement of the static class hierarchy that we developed for the first object-oriented version.
- The identification of the precise set of *listeners* in the Input Forms and a detailed specification of their patterns of interaction with other listeners/objects.
- The encoding of the framework. There was a considerable change in the Java code for the first version and this second version because we moved from Java Development Kit 1.0 to 1.1 in the second version and event-handling was done in a totally different manner in 1.1 than in 1.0.
- A thorough debugging of the framework, using the Radiotherapy Input Form as a testing instance. The Radiotherapy form has a Patient Select, Patient Entry, Radiotherapy Select and Radiotherapy Entry screen. This form was chosen for testing because it was relatively small, yet it exhibited a full range of input-form behavior, i.e., it required inclusion of all the listeners.

After the framework was complete, the GUI portion of the four input forms described above was fully implemented and completely debugged in one person-week, although the coding of detailed interactions of the form with the database took considerably longer. The new users, generating the Chemotherapy form, estimated that it required between 1.5 to 2 hours to complete this exercise, again producing completely debugged graphical-user interfaces.

Based on this experience, we feel fairly confident in making the following observations.

- The time to develop the five Input Forms current in our project would be approximately the same order of magnitude using a procedural approach, an object-oriented approach, with emphasis on static reuse components, or a framework approach, i.e., about 30 person-weeks.
- Now that the framework is completed, as we require more input forms, for example for researchers, the generation of the GUI components of new applications using the framework will require very little time, on the order of magnitude of several hours. With other approaches, each new form would require many person-weeks, depending upon size and complexity.

- The *quality* of new applications that are generated using the framework is extremely high. The GUI component of new Input Forms, generated both by developers and by users who were not involved in framework development, are virtually without any bugs.

8 Conclusions

We have been supporting data input into the Neuro-Oncology Information System by way of graphical user interfaces for many years. We first implemented the Input Forms using procedural methods, then moved to an object-oriented approach, with an emphasis on static reuse of components. Finally, we progressed to using a framework design in which we encoded explicitly in the components their common patterns of interaction.

Metrics have yet to be defined that are suitable for evaluating a framework design, so we offered anecdotal evidence about our success with a framework. The technical and nontechnical factors that impact reuse merged in our project to produce virtually 100% reuse and virtually bug-free applications when we generated new Input Forms using the framework.

In this particular project, any additional time that was required to generate a framework for Input Forms was far outweighed by the anticipated decrease in time to generate new applications and, more importantly, by the significant increase in the quality of the new applications.

Our framework is focused almost exclusively on the graphical user interface portion of the Input Forms, including the basic connectivity between the form and the database. However, the framework offers no assistance in generating the actual database views that are required to support each Select/Entry screen pair, nor in generating the often complex code that is required to update or create records in the database when the user hits the Ok button on an Entry screen. These latter tasks remain demanding in development and debugging time and are being addressed in an ongoing, separate effort to generalize and abstract behavior.

References

[1] Robert G. Fichman and Chris F. Kemerer. Object technology and reuse: Lessons from early adopters. *Computer*, pages 47–58, October 1997.

[2] Sandra A. Mamrak, John Boyd, and Iván Ordó nez. Building an information system for collaborative researchers. *Software Practice and Experience*, 27(3):253–263, March 1997.

[3] Department of Computer and Information Science, The Ohio State University, Columbus, Ohio. *Neuro-Oncology Information System Technical Reference*, June 1996. Also found at http://www.acuity.cis.ohio-state.edu.

[4] Oracle Corporation, Redwood City, CA. *ORACLE7 Server Application Developer's Guide*, December 1992. PN 6695–70–1292.

[5] Oracle Corporation, Redwood City, CA. *Oracle Data Browser User's Guide*, October 1992. (Version 1.0) PN 6480–10–1092.

[6] Oracle Corporation, Redwood City, CA. *Oracle Forms Reference Manual Volume 1*, July 1993. (Version 4.0) PN A11988–1.

[7] Oracle Corporation, Redwood City, CA. *Oracle Forms Reference Manual Volume 2*, July 1993. (Version 4.0) PN A11989–1.

[8] Oracle Corporation, Redwood City, CA. *SQL*Net Administrator's Guide*, July 1993. (Version 2.0) PN A11325–1.

[9] David H. Friedel Jr. and Anthony Potts. *Java Programming Language Handbook*. Coriolis Group Books, 1996.

[10] Jamie Jaworski. *Java Developer's Guide*. Sams Net, 1996.

[11] Brent B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall PTR, New Jersy, 1995.

[12] Valerie Quericia and Tim O'Reilly. *X Window System User's Guide.* O'Reilly & Associates, Inc, 1991.

[13] Open Software Foundation. *Application Environment Specification (AES) User Environment Volume.* Open Software Foundation, 1990.

[14] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[15] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[16] Shishir Gundavaram. *CGI Programming on the World Wide Web.* O'Reilly & Associates, Inc, 1996.

[17] Grady Booch. *Object-Oriented Analysis and Design.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[18] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[19] Peter Pin-Shan Chen. The Entity–Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[20] Fernando Brita e Abreu and Rogerio Carapuca. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems Software*, 26:87–96, 1994.

[21] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[22] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.