A Flexible Browsing Framework for the Morpheus Transform Repository

by

Tiffany Dohzen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

August 21, 2007

Certified by

Michael Stonebraker
Adjunct Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Abstract

This thesis describes the design and implementation of a flexible browsing framework for the Morpheus project. The goal of the Morpheus project is to address the problem of heterogeneous information integration by providing a central repository for database transforms and datatypes, and storing metadata associated with each object, such as author, name, description, etc. One of the central features of Morpheus is the ability to explore the repository graphically by using this associated metadata. The framework described in this thesis provides an extensible interface for searching and browsing the repository visually, and comprises both back-end components that integrate with the repository as well as a full graphical user interface.

The framework is designed to seamlessly integrate browsing and searching by means of user-defined search filters that restrict which results are displayed. We also introduce the concept of a named query, an object that associates a set of filters with a name. This allows the same set of results to be displayed in different ways by different browsers. Finally, three example browser implementations are presented to demonstrate the framework in use.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor
Acknowledgments

First and foremost, I must thank my thesis supervisor, Dr. Michael Stonebraker, for his infinite patience and support in completing this thesis, even when I was not the most diligent of students.

This thesis also would not have been possible without the hard work of my fellow Morpheus project colleagues: Mujde Pamuk at MIT, and the team at the University of Florida, led by the capable Drs. Joachim Hammer and Peter Dobbins.

To my close friends at MIT: the cool Davis kidz, the Techsters, girls-gone-six, and most of all, my AXΩ sisters — thank you for all of your love, support, and friendship. I wouldn’t have made it through MIT without you.

Finally, this thesis is dedicated to my mother, for all that she has given me and continues to give me in life.
## Contents

1 Introduction 13

2 Morpheus Overview 15
   2.1 Architecture 15
   2.2 Dimensional Browsing 16
   2.3 Morpheus 1.0 Browser 17
      2.3.1 Limitations of Morpheus 1.0 20

3 Browsing Framework 21
   3.1 Overview 21
   3.2 Filters 21
   3.3 Named Queries 22
      3.3.1 Predicates 22
      3.3.2 Object Retrieval 23
      3.3.3 Query Operations 25
   3.4 Data Browser 25

4 The User Interface 27
   4.1 Overview 27
   4.2 Top-level controller 28
   4.3 Data browser 30
   4.4 Named query panel 31
   4.5 Filter panel 33
4.6 Further Development ........................................ 42

5 Browser Implementations .................................. 43
  5.1 Simple Text Browser ..................................... 43
  5.2 Prefuse Toolkit .......................................... 44
  5.3 Domain Browser ......................................... 46
    5.3.1 Description ......................................... 46
    5.3.2 Implementation Details .............................. 49
  5.4 Lineage Browser ......................................... 51
    5.4.1 Description ......................................... 51
    5.4.2 Implementation Details .............................. 52

6 Related Work ................................................ 55
  6.1 Multidimensional Browsing .............................. 55
    6.1.1 Visual Queries ..................................... 56
  6.2 Hierarchical Visualization ............................. 56
    6.2.1 Context ............................................. 57
    6.2.2 Zooming ............................................ 57

7 Conclusions and Future Work .............................. 59
  7.1 Core Framework Improvements .......................... 59
  7.2 Browser Implementations ................................ 60
  7.3 User Tests and UI Improvements ....................... 61

A NamedQuery API .......................................... 63
  A.1 Nested Enumerations .................................... 63
    A.1.1 Parameter .......................................... 63
    A.1.2 Type ................................................ 63
  A.2 Nested Class ........................................... 65
    A.2.1 Predicate .......................................... 65
  A.3 Key Methods ............................................ 65
    A.3.1 Constructors ....................................... 65
List of Figures

2-1 High-level overview of Morpheus architecture .......................... 16
2-2 The Morpheus 1.0 browser interface: keyword search .................. 18
2-3 The Morpheus 1.0 browser interface: category browsing .............. 19

4-1 Class structure of primary UI components ............................... 28
4-2 Initializing browsing or searching from within Morpheus .............. 29
4-3 The basic browser user interface within Morpheus ..................... 30
4-4 High-level browser control flow .................................... 31
4-5 The query operations panel ........................................ 32
4-6 Saving a named query as ‘mit’ ...................................... 33
4-7 Creating a new query ................................................ 34
4-8 Preparing to take the union of the ‘mit’ query and the current (‘dev’) query .................................................. 35
4-9 The new, unnamed query created from the union of the ‘mit’ and ‘dev’ queries .................................................. 36
4-10 The filter panel .................................................... 37
4-11 The SQL tab ...................................................... 38
4-12 The cost filter control ............................................ 39
4-13 The type filter control ............................................ 40
4-14 The domain filter control button .................................. 40
4-15 The simple browser showing mixed type results ...................... 41
4-16 The domain filter control window with the MIT category in focus .. 41
4-17 A listing of currently applied filters, with options to delete .......... 42
5-1 The simple browser, displaying search results in a textual listing...

5-2 The simple browser with the first result selected as the current Morpheus object.

5-3 The architecture of the browsing framework for Prefuse-integrated browsers

5-4 The domain browser automatically selecting the 'MIT' category based on the selected Morpheus object.

5-5 The domain browser with incremental search results displayed.

5-6 The domain browser with matching categories highlighted.

5-7 The lineage browser with matching objects highlighted.
Chapter 1

Introduction

One of the major issues facing large organizations today is the aggregation of large volumes of data stored in disparate databases. With the rise of data warehouses as central repositories for data, there has been an increasing need to efficiently collect, process, and store data coming from these various sources. Further, there is also an increasing demand for data integration across different domains—for example, in interdisciplinary fields such as computational biology.

Various approaches, from schema matching [9] to standardized ontologies for text matching [20], have been used to address the information integration problem in the past. The approach taken by the Morpheus project [10] is to build a browseable, searchable repository of reusable database transforms. This approach combines a high-level transform construction toolkit (TCT) with an easily navigable browsing interface to locate transforms in the repository.

The motivation behind such a repository is the premise that there is a large amount of commonality between many transforms—different currency conversions, for example, or transforms between different date formats. We can encapsulate metadata about a transform in a central repository, from its input and output data types to a textual description describing its function. Each of these metadata fields we store forms a new metadata dimension that we can then use to browse the repository; we can also use these dimensions to search for a particular class of transforms. As the repository grows, it will become easier to find a transform that provides a particular
desired functionality. Even if the exact transform desired is not found, the TCT allows users to easily modify existing transforms and create their own.

One of the key components of the Morpheus project is the browsing and searching interface. Because the primary goal of the repository is to encapsulate metadata about various transforms and data types, it follows logically that users should be able to use this metadata to quickly and efficiently locate a desired object or class of objects, as well as wander the repository by browsing via dimensions corresponding to assorted metadata fields. The underlying framework for the browsing and searching interface, as well as an implementation utilizing this framework, is the focus of this thesis.
Chapter 2

Morpheus Overview

2.1 Architecture

A high-level overview of the basic Morpheus architecture is shown in Figure 2-1. There are three main components to Morpheus: the backend repository for storing the transforms, a graphical interface for browsing and searching the repository, and a high-level transform construction toolkit (TCT) to allow users to create new transforms and add them to the repository or to modify existing transforms. All of these components are tied together with a top-level user interface that runs as a standalone Java application.

The primary back-end component is the POSTGRES database, which serves as both repository and platform for transform execution. Both transforms and data types are registered in the database with accompanying metadata, including name, creation date, author, keywords, etc. All transforms are registered as POSTGRES functions, and all data types are registered as user-defined POSTGRES data types, allowing users to directly load data and execute transforms within Morpheus itself. In this sense, Morpheus acts as a self-contained extract-transform-load (ETL) tool.

The TCT is a drag-and-drop tool that can be used to create a boxes-and-arrows diagram of a transform. This diagram is then compiled into a Java function and registered in the repository with user-specified metadata as well as an XML description that corresponds to the diagram itself. The tool includes transform primitives such
as addition, multiplication by a constant, table lookup, etc.

In Morpheus 2.0, the most recent incarnation of the project, each core feature (object creation, browsing, execution, etc.) has an internal window, or Morpheus screen within the main Morpheus window. To facilitate shared data and communication across these Morpheus screens, the system has the concept of a currently selected object, which represents the transform or data type object that is currently "active". This object can be accessed and written by all of the relevant Morpheus screens, and must be of type MorpheusObject, an interface available to all Morpheus screens.

### 2.2 Dimensional Browsing

When a transform or data type object is registered in the repository, a set of metadata fields are associated with that object. These fields function as dimensions along which to browse the repository. Each of these dimensions serves two purposes: first, as a primary dimension along which to browse, and secondly, as a filtering dimension to restrict displayed results. For more detail on filters, see Section 3.2.
One such dimension is the general-purpose classification to which a given repository object belongs, which we call a domain. Domains are hierarchical in nature; top level domains might include Travel and Education, while lower levels would have domains encompassing objects related to a specific airline or university, for example. Another hierarchical dimension might be author, with individual authors at the bottom of the hierarchy, and organizations of authors in the form of labs, departments, schools, companies, etc. serving as intermediate nodes in the hierarchy.

There are also non-hierarchical dimensions along which to browse the information space. The input and output data type fields associated with a transform are two such examples—when viewing a transform, a user can browse to the input or output data type. Some non-hierarchical dimensions, such as cost, additionally have a natural ordering that will be reflected in the navigation.

2.3 Morpheus 1.0 Browser

Figure 2-2 shows a screenshot of the Morpheus 1.0 browser with a keyword search performed for transforms that start with 'mit'. This browser allows entry into the multidimensional metadata space via keyword search, with results displayed as a text listing based on relevancy to the search terms. Users can choose which object types to search via the right sidebar buttons, or choose to search all types simultaneously.

Also implemented in the 1.0 browser is a primitive domain browser, via a simple tree visualization, as shown in Figure 2-3. Clicking on the name of a domain effectively does a repository search for all objects that fall in that domain, and returns them in a text listing in the same format as for keyword search.

Users then can select a particular result (repository object) as their focus from the list of results and perform other Morpheus functions relating to the selected object. For example, selecting an object and then going to the TCT screen allows users to view and modify the structure of that particular object, if applicable.
Figure 2-2: The Morpheus 1.0 browser interface: keyword search
Figure 2-3: The Morpheus 1.0 browser interface: category browsing
2.3.1 Limitations of Morpheus 1.0

Although Morpheus 1.0 served its purpose as a proof-of-concept repository and transform tool, there are several limitations in the design and implementation, particularly with the browsing and searching interface. As the basic hierarchical category browser was developed with a priority on getting a working implementation, rather than building a flexible, extensible framework, there is no way to easily add a new browsing mode or visualization. Further, the design of the overall browsing interface limits the ways in which the actual act of browsing can occur: users only have the option of browsing or searching by keyword, but not both.

The design of the keyword search panel on the right of the browsing screen also presents a restriction on searching functionality, as it only allows users to search for a single type, or for all types at once. Additionally, the use of a new button for each type unnecessarily clutters the interface, and imposes an upper limit on the number of types that can be added before the lack of screen real estate becomes a problem.

All of the above issues were taken into consideration when designing the browsing framework that is the focus of this thesis. Specifically, the lessons learned from the Morpheus 1.0 browser led to the following design goals for the new browsing framework:

- Full, seamless integration between search and browse functionality
- An easily extensible browsing framework to allow for future development of different browsing modes and visualizations
- The ability to restrict results by multiple dimensions at once
- A clean, uncluttered user interface for searching and browsing
- Retention of existing browsing functionality and integration with other Morpheus functions
Chapter 3

Browsing Framework

This chapter describes the design and implementation of the general browsing framework. The focus of this chapter is the core framework structure; for details on the user interface and its integration with this framework, see Chapter 4.

3.1 Overview

The browsing framework was designed to provide a flexible interface to allow for easy integration of future development, while maximizing code re-use. Conceptually, the browsing framework consists of three basic components: a data browser, which encompasses the visual representation of repository objects and reflects the current browsing state; a set of filters defined by the user to restrict which repository objects are in view at any given time; and a set of named queries, also defined by the user, which represent a saved set of filters and, indirectly, the corresponding objects that match these filters. Each of these components plays a key part in the core browsing functionality, and is described in more detail in the sections below.

3.2 Filters

Filters play a key role in the browsing experience; a large fraction of the time spent browsing is spent defining, applying, and removing filters. Like the traditional use of
filters in search interfaces [14], filters in Morpheus allow users to restrict the set of
displayed objects by performing search queries along multiple dimensions. Multiple
filters can be applied at once to a named query, and each named query keeps track
of what filters have been applied, so that as the repository is updated, the resulting
set of objects associated with a particular query are automatically updated without
the need for users to redefine or reapply any filters.

3.3 Named Queries

A named query is the underlying data structure for the browsing framework, and is
implemented in theNamedQuery class. It represents a set of filters on the repository,
much like a SQL query to a generic database. Like a SQL query, a named query is not
only able to store a set of filters (called predicates in the implementation), but also
provides methods to allow conjunctions and disjunctions of existing named queries to
produce new named queries. Further, the “named” part of the named query stems
from the goal of providing a mechanism to define a set of queries that produce the
desired results, save such a query to memory using a user-defined name, and re-load
the query without having to redefine each filter. The most salient components of
theNamedQuery structure are described below; a more detailed API is available in
Appendix A.

3.3.1 Predicates

A predicate is the core component of a named query, and is the concrete representation
of the conceptual filter. Each predicate has three subcomponents: a Parameter,
which indicates the metadata dimension in the repository that this predicate applies
to (name, author, description, etc.); a Type, which describes how to restrict the
object set; and a Value, which is the string value of the predicate. The predicate
Type controls how the Value is used in the search for matching objects, and is a loose
indicator of how restrictive a particular predicate is (exact match, within a range,
etc).
Both Parameter and Type are implemented using Java's typesafe enum construct, which provides several benefits, including: true type safety, rather than dependence on overloaded primitive types such as integers; the ability to include class-like structures such as methods and fields, but without the overhead of a full-blown class; and the flexibility to easily add future values without affecting how existing code works. For a full description of the enumeration values for Parameter and Type, see Appendix A.1.

Examples of predicates:

- **NAME = 'mit',** which selects all repository objects named 'mit'

- **DATE LESS THAN '1/1/2007',** which selects all objects with a creation date before Jan. 1, 2007

- **AUTHOR = 'pete*',** which selects all objects who have an author prefixed by 'pete'

To prevent thread-safety issues, all methods which modify a named query, including adding predicates, removing predicates, and changing the name of a query, are synchronized using Java's built-in synchronization mechanisms.

### 3.3.2 Object Retrieval

All NamedQuery objects support a `getObjects()` method, which retrieves matching objects from the database. There are two primary search mechanisms for matching objects: a direct SQL query to the database, using SQL "where" clauses derived from predicate values, and a keyword-based indexing and search engine provided by the Lucene library [1].

When Morpheus starts up, a utility class retrieves all database objects and writes a select set of parameters to a set of textual index files as specified by the Lucene interface. All text-based parameters, including author name, object name, and object description, are indexed and searched by Lucene. Additionally, a unique ID assigned by the repository to each object is indexed; this allows the system to retrieve further information about each object beyond the indexed text-based parameters. Another
utility class provides methods to search the Lucene index in the context of these parameters.

At a high level, the algorithm for locating matching objects consists of the following:

1. Check if there is a predicate for the TYPE parameter, restricting results to only transforms, data types, etc. If so, set appropriate internal variables so that the correct database tables are searched, etc.

2. For each predicate associated with the current query, assign the predicate to be Lucene-searched or database-searched

3. Use the Lucene search engine to retrieve the unique object IDs of all repository objects matching the keyword search filters

4. Using these object IDs, retrieve enough information about each object from the database to generate a set of MorpheusObject results; store these results locally

5. For the remaining (non-keyword) predicates, including any SQL predicates, generate a SQL query to retrieve the appropriate information from the database

6. Execute the query to retrieve all objects matching the non-keyword predicates; store these results locally

7. Combine the Lucene and database results as follows: prioritize results in the intersection of both result sets, followed by Lucene-only results, followed by database-only results

8. Fetch any additional metadata (e.g. domain information, which is not retrieved by default) requested by the calling method from the database and add this data to the combined result set

9. Return this combined, prioritized set of results as an array of MorpheusObjects
### 3.3.3 Query Operations

The framework supports a number of named query operations, including the ability to save named queries under a particular name, load a previously saved query, retrieve all repository objects that match a query’s predicates, and combine two queries via conjunction or disjunction.

To increase modularity, the `NamedQuery` class makes use of the *Observer* or *Listener* design pattern [11]. Listeners are registered with a particular `NamedQuery` object, and events are fired whenever the named query is updated. All listeners must conform to the `NamedQueryListener` interface by implementing the `queryUpdated()` method, which is called when an event is fired.

The primary use of listeners is to force UI components to repaint themselves as needed whenever a query is updated, but the `NamedQueryListener` interface is not UI-specific; it is feasible that in future versions of Morpheus, components that are unrelated to the user-facing GUI will also implement `NamedQueryListener`.

Once a listener has been registered with a specific named query, it is notified whenever that query is updated. This allows different UI components to be easily decoupled from each other, as the only shared state necessary is the named query itself. The listener interface also eliminates the need for components to poll query objects periodically to check for changed state, and ensures that updates to the dependent components, which often involve expensive operations such as database queries and screen repainting, only happen when necessary. Listeners can be registered and deregistered at any time. To prevent thread-safety issues, listener operations are also synchronized using Java’s built-in synchronization mechanisms.

### 3.4 Data Browser

The data browser is a component that has three primary functions:

- Retrieve and display results from the database whenever the named query associated with this data browser is updated
- Provide visualization-specific controls for navigation and general user interaction

- Interface with the rest of the Morpheus application via the stored MorpheusObject

The underlying purpose of the data browser is to control the visual display and navigation of repository objects in a specific format or visualization. To maximize flexibility, how these objects are displayed is completely transparent to the rest of the framework via a clearly-defined interface between the data browser and the rest of the back-end.

This modularity allows for different types of data browsers, where a type of data browser is simply a particular visualization technique for displaying objects. One type of data browser might be a basic text listing in the manner of traditional search engines, for example. Another more sophisticated browser might display data in the form of a standard graph or tree visualization; yet another might use treemaps or a timeline. To minimize duplicate code, an abstract DataBrowser class contains a skeletal implementation of common browsing functionality; all specific data browser types inherit from this abstract class and only override the methods necessary to provide custom visualizations and display functionality. Further, since multiple dimensions may use the same type of visualization — a graph visualization might be used to browse both by category and lineage, for example — the data browsers are designed to be as dimension-independent as possible.
Chapter 4

The User Interface

This chapter describes the design of the user interface and its interaction with the rest of the browsing framework and the Morpheus application.

4.1 Overview

The components of the browsing framework are designed to support a flexible and modular user interface. There are four core UI components: the top-level browser screen, which serves as the main UI controller; the data browser, responsible for data display and navigation; the filter panel, which allows users to set criteria that displayed data must meet; and the named query panel, which provides information about the currently active named query and controls to perform operations on different named queries. Together with the back-end structures discussed in Chapter 3, these UI components provide the core browsing functionality for the system. The relationships between functionality, front-end components, and back-end data structures is shown in Table 4.1.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Back-end structure</th>
<th>UI structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filtering</td>
<td>Named queries</td>
<td>Filter panel</td>
</tr>
<tr>
<td>Query operations</td>
<td>Named queries/Morpheus control layer</td>
<td>Named query panel</td>
</tr>
<tr>
<td>Visual browsing</td>
<td>Viz-specific data structures</td>
<td>Data browser</td>
</tr>
</tbody>
</table>

Table 4.1: Browsing functions and corresponding front- and back-end structures
Because one of the goals in designing this framework was to allow for flexibility in future development, modularity and code re-use was highly prioritized. Figure 4-1 shows the class structure and relationships between the core UI component classes. Note that there are two abstract classes, BrowserScreen and DataBrowser, which contain code common across different data browsers and browser screens. Any future browser screens and/or data browsers that are developed for this framework in the future should inherit from the appropriate abstract class to reduce code duplication. Although this makes the class structure slightly more complex, this design ensures more consistent behavior between classes while still allowing for substantial flexibility in the actual implementation of a data browser or browser screen.

### 4.2 Top-level controller

The top level UI controller, known as a *browser screen*, is the link between the primary Morpheus controller and the browsing framework. This is the screen that is initialized by Morpheus when the user first selects one of the “Browser” menu options from the top-level Morpheus screen (see Figure 4-2). As with all other Morpheus screens, browser screens function as internal frames within the main Morpheus application. Figure 4-3 shows a browser screen, in this case the simple (textual) browser screen, within the context of the main Morpheus application, before any filters have been applied. There are three major components within the browser screen: the data browser, the query panel, and the filter panel. The data browser is the largest of the
components and takes up the majority of the browser screen. The vertical panel on the right is the filter panel, and the panel on the bottom left portion of the screen is the query panel. These components are described in more detail below.

The browser screen initializes the three subcomponents (data browser, query panel, and filter panel) and sets up any data structures needed by a particular data browser; the query panel and filter panel remain unchanged. The browser screen is also responsible for maintaining shared state, most notably a NamedQuery object that represents the current query, the query that is currently active in the browser. Finally, to ensure that all updates are reflected accordingly across the different components, the browser screen registers listeners for the components with the current query.

Browser screens are, in general, associated with a specific dimension: there is a DomainBrowserScreen for exploring the repository based on how objects are categorized, for example. Keeping this association at the browser screen level serves two purposes: first, it means data browsers can be dimension-independent and thus re-used by different browser screens and different dimensions; second, it allows for other UI components, such as the filter panel, to be customized based on the associated dimension. The single exception to this dimension-specific association is the SimpleBrowserScreen. This browser screen is designed to be used with the TextualDataBrowser, which provides a text-only listing of results sorted by relevance to the applied filters. Since this listing does not provide any true browsing capability, there is no need for the SimpleBrowserScreen to be associated with a particular dimension.
4.3 Data browser

As described in Section 3.4, the data browser is conceptually responsible for providing visualization-specific browsing capabilities. A high-level overview of the control flow from user interactions to displayed results is shown in Figure 4-4. Note that there are three main ways in which the user can interact with the system. The top two, representing query operations and filter addition/removal, trigger an update to the current query, which in turn triggers callbacks to the listeners associated with each of the main components. For the data browser, this callback usually involves a data fetch from the database to obtain the most recent results that match the criteria specified by the filters. These results are then displayed in the data browser's panel.

The last type of user interaction involves data browser-specific controls. This might consist of clicking on nodes in a tree, or zooming in or out of the display. Unlike user interaction with the filters or query operations, these data browser interactions do not necessarily require a call to the database; data may be cached locally to
provide faster performance. Also, every data browser provides some mechanism for interacting with the current MorpheusObject, an object that is global not only across all browsing screens, but across the entire Morpheus system. This allows an object selected in a browsing screen to be edited in the creation tool, or executed in the local execute screen, for example.

4.4 Named query panel

The named query panel provides the name of the current query and UI controls for various query operations. These include:

- **Save query**: Save the current query by giving it a name and adding it to the set of saved queries. Currently, these are saved in memory and thus are only available for a single session, but in the future will be saved to disk and retrievable across sessions. Saved queries are stored using the top-level Morpheus controller, known as the Control Layer, and are therefore available to other parts of the application, not just the browsing framework. Once a query has been named, any changes made to it will automatically be saved, as long as it is the current query.

- **Load query**: Load a previously saved query by setting the current query to be the specified query; if the previous current query was not saved, any changes
the user made to that query will be lost.

- **New query**: Create a new, unnamed, empty query and make it the current query. As filter changes are automatically saved to named queries, this option must be selected before performing *any* filter operations that are intended to be applied to a new query.

- **Union**: Create a new, unnamed query that is the result of taking the union of the current query and the specified saved query. This unnamed query is then set as the current query.

- **Intersection**: Create a new, unnamed query that is the result of taking the intersection of the current query and the specified saved query. This unnamed query is then set as the current query.

These options are presented in a drop-down control, as shown in Figure 4-5. The currently selected query’s name is also displayed to the user, and depending on what query operation is selected, anywhere from zero to two additional drop-down controls are dynamically presented to the user to select other named queries for the operation. No actions are taken until the user clicks on the “Go” button, which modifies the query objects as necessary and repaints the screen to reflect the updates.

In the example shown in Figure 4-6, the user is naming a set of results in the simple browser screen as ‘mit’. The currently active filters are then saved as a new NamedQuery object with the name ‘mit’ in the top-level Morpheus ControlLayer, and for the remainder of the session, this saved query will be accessible from any browser screen. Any filter modifications from this point will now automatically be applied to and saved with the ‘mit’ named query, as long as it is the current query.
To make a different query be active, an existing query must be loaded, or a new query either explicitly created with the ‘New Query’ operation (Figure 4-7) or implicitly created with the Union or Intersection operations. Figures 4-8 and 4-9 show the results in the simple browser immediately before and after taking the union of two queries: the ‘mit’ query, which filters for objects in the ‘MIT’ category, and the current query, called ‘dev’, which filters for objects created by ‘dev’. Changing which named query is currently active causes any listeners registered to the previously active query to be re-registered with the newly active query, and also triggers an update notification to all listeners so that any displayed data is up-to-date.

4.5 Filter panel

The filter panel provides UI controls to apply filters to the current named query. As shown in Figure 4-10, the filter panel is split into two tabs: the Basic tab, and the SQL tab. The Basic tab contains traditional form controls to allow users to
dynamically create filters to apply, while the SQL tab serves as a mechanism for more advanced users who wish to create more complex filters. The SQL tab, shown in Figure 4-11, provides a text box for users to restrict the result set by entering a SQL "where" clause. The fields available for use in the clause are listed below the text box; for security and usability reasons, these do not directly correspond to actual database fields, but rather serve as placeholder fields, which are then translated to the appropriate corresponding database field names on-the-fly as results are retrieved.

The Basic tab is split into two subsections. The first section, labeled "Keyword Search", allows users to perform a keyword search on text-based parameters that have been indexed by the Lucene indexing and search engine. The second section, labeled "Other filters", provides controls for users to enter restrictions on non-text-based parameters. These predicates are translated directly to SQL restrictions when querying the database for matching objects. The combo box labeled "Parameter" controls which of these parameters is visible at any given time; selecting a different
Figure 4-8: Preparing to take the union of the ‘mit’ query and the current (‘dev’) query
Figure 4-9: The new, unnamed query created from the union of the 'mit' and 'dev' queries
Figure 4-10: The filter panel
Basic SQL

Enter SQL where clause:

```sql
where AUTHOR='dev'
```

The following fields are available:
- NAME
- AUTHOR
- DESCRIPTION
- DATE
- COST
- TYPE
- DOMAIN

Add filter(s)

Current filters

Figure 4-11: The SQL tab
Figure 4-12: The cost filter control

parameter will change the type of control visible.

Currently, there are controls for three parameters implemented: object cost (the size of the object’s binary representation in the database) in KB (Figure 4-12); type of object (Figure 4-13); and domain (Figure 4-14). The implementation of the cost filter is not yet complete; currently only a maximum or a minimum cost, but not both, may be applied to a named query. The type filter control includes an option for ‘no restriction’, which opens the search to all repository objects (see Figure 4-15). This is the default option; re-selecting ‘no restriction’ after adding a type filter effectively removes the filter.

The domain control brings up a separate window (Figure 4-16) that contains a smaller, more limited version of the domain browser screen described in the next chapter (see Section 5.3). This window allows users to dynamically explore the domain hierarchy by clicking on domains to view sub-domains. Double-clicking on a domain restricts the displayed results to those contained in that domain.

Up to two filters may be added at any time: a single SQL filter, or a keyword search filter and/or other filter. Clicking on the “Add filter(s)” button triggers an update to the current query, which in turn triggers notifications to the appropriate listeners as shown in Figure 4-4. One of these listeners is responsible for updating the list of filters located at the bottom of the filter panel, where all of the filters for
Figure 4-13: The type filter control

Figure 4-14: The domain filter control button
Figure 4-15: The simple browser showing mixed type results

Figure 4-16: The domain filter control window with the MIT category in focus
the current query are displayed, along with the option to delete some or all of them (Figure 4-17). This listing of filters persists regardless of which tab, Basic or SQL, is currently active.

4.6 Further Development

Of the components described above, the data browsers are the most amenable to further development, as they are the most decoupled from the other UI components. However, all of the UI components, from the filter panel to the browser screen, are fairly extensible, as they are all encapsulated in their own class and have minimal direct interaction with each other.
Chapter 5

Browser Implementations

As a proof of concept, three data browsers have been developed that utilize the browsing framework described in Chapter 3. This chapter described the design and implementation of these data browsers. The first, the textual browser, is designed to emulate the functionality of the Morpheus 1.0 searching interface; the others utilize the Prefuse visualization toolkit [13] to provide a richer, more sophisticated browsing experience. See Section 5.2 for details on integration with Prefuse.

5.1 Simple Text Browser

The simplest implementation of the browsing framework is the SimpleBrowserScreen, which utilizes a TextualDataBrowser. These provide a simple textual listing of search results, similar to that of Morpheus 1.0. Figure 5-1 shows the simple browser displaying matching results for a named query with two filters: one on the object name, using the wildcard * to search for names with a prefix of ‘mit’, and a type filter to only search for transforms. The displayed results show the type, name, author, and description of each matching result.

The radio button next to each result in the display allows users to select one of the currently displayed results as the current Morpheus object. As discussed earlier, this allows other browser screens and other Morpheus application screens to access an object selected in this browser screen and view its properties. In Figure 5-2, the
first object has been selected by the user. In the bottom-left corner of the Morpheus screen, this is indicated by the text “transform: mittoufstudent”. This text will persist regardless of which Morpheus screens are open.

This implementation relies heavily on the NamedQuery API to provide the appropriate data to display. As there is no distinct visualization for this browser, the TextualDataBrowser simply displays the results returned from the getObjects() method of the current named query, in the order that they are returned.

5.2 Prefuse Toolkit

Both of the following browsers make use of the Prefuse visualization toolkit. Prefuse, an open-source 2D Java visualization aid, is a polyolithic toolkit, comprised of many interdependent but separate components. It generally follows the model-view-controller user interface design pattern, with separate APIs for backing data structures (model), visual abstractions (view), and user interaction (controller). The polyolithic nature of
Figure 5-2: The simple browser with the first result selected as the current Morpheus object
Prefuse allows for fine-grained control over interactions, while the many pre-coded components allows for visualizations to be created relatively quickly.

Figure 5-3 gives a general idea of the overall architecture of these browsers in the context of the entire browsing framework. The use of Prefuse code is almost completely contained within the data browser classes, although the browser screens do make use of Prefuse data structures to initialize certain data browsers. This diagram also shows the layers of interaction within the browser screen level: at the bottom is the actual database, which is only accessed directly by the NamedQuery class via the JDBC API. The current named query object serves as an intermediate layer, and its API allows the top-level components (the actual UI panels) to send updates to the current query, receive notifications of changes, and in the case of the data browsers, request matching results from the repository. User-provided input and user-visible output are also distinguished to emphasize the distinction between front-end UI components and back-end data structures.

5.3 Domain Browser

The domain browser allows users to view repository objects grouped by domain. Domains are displayed as nodes of a tree; the tree structure is determined by a preset hierarchy of domains stored in the database. The root of the tree is the top-level domain, called “All”.

5.3.1 Description

When the domain browser is initialized, it attempts to retrieve the domain of the currently selected Morpheus object. This domain becomes the domain in focus: it is highlighted in the display, and the path back to the root domain is expanded. If there is no currently selected object, or no such category can be found, the root domain is in focus instead.

As an example, Figure 5-4 shows what the domain browser initially looks like with a transform from the MIT category (‘mitoufstudent’) selected as the current
Figure 5-3: The architecture of the browsing framework for Prefuse-integrated browsers
Figure 5-4: The domain browser automatically selecting the 'MIT' category based on the selected Morpheus object

Morpheus object, as indicated in the bottom left corner of the screen. The browser automatically expands to the MIT domain node, and indicates it is the domain in focus by painting the node green. Note that the path from the currently selected domain back to the root is also highlighted in a darker shade of green to give the user a quick sense of the domain hierarchy.

Figure 5-5 shows the incremental search feature of the domain browser. The currently selected node, indicated in green, is the root node; the user has performed a search for domains with the prefix 'fin'. As each letter is typed, any domains that match the prefix typed in so far are highlighted in red.

To limit the number of domains that are displayed at any given time, the browser only displays two levels of children from the domain currently in focus, as well as the full path back to the root. Clicking once on a domain selects it and forces a repaint to display that domain's children. Double-clicking on a domain adds that domain to the metadata associated with the currently selected Morpheus object; if no object
is currently selected, it creates a new object with the domain as the sole property. This information can then be used by other Morpheus screens; for example, the TCT screen might use this feature to associate a newly created transform with the correct category.

When filters are applied to the current query, the domain browser retrieves the domains of matching repository objects. These matching domains are bolded in the display, while non-matching domains are grayed out. In Figure 5-6, the MIT domain is in bold text, indicating that this domain contains objects that match the filter criteria (author='dev', name='mit*'), while the non-matching domains are grayed out.

5.3.2 Implementation Details

As with all browsing screens, the domain browser consists of two components: the top-level browsing screen, and the data browser, which is responsible for the actual data visualization. The domain hierarchy data is pulled directly from the database.
Figure 5-6: The domain browser with matching categories highlighted by the DomainBrowserScreen and used to initialize the Prefuse tree structure, which is then passed to the CategoryDataBrowser. This tree initialization is done at the browser screen level rather than the data browser level so that in the future, other browser screens that require a category/tree-like visualization can easily reuse the CategoryDataBrowser code by passing it a tree initialized with different data.

The CategoryDataBrowser, which contains virtually all of the actual visualization code, is implemented using a modified version of the TreeView demo code from the Prefuse toolkit. The modifications included the following customizations:

- Changing the backing data source to correspond to the tree structure passed in during construction
- Changing the node coloring action to gray out and bold node labels based on which domains match the filter criteria of the current query
- Adding an initial check for a set Morpheus object and if so, setting the corresponding domain as the initial focus
• Adding a category selection action to set the current Morpheus object based on the domain in focus

• Replacing the right-click zoom control with a mouse wheel zoom control

• Removing orientation-related actions, as they are not used by the data browser

• Adding a double-click control to trigger the new category selection action

Additionally, when the data browser is notified of an update to the current named query, it requests matching objects from the repository by calling the `getObjects` method of `NamedQuery` with an additional parameter representing the `DOMAIN` parameter type. This signifies that in addition to the standard object information that is returned from `getObjects`, an additional metadata field should be set with the returned repository objects to store domain information. This information is then used to determine which categories are a match to the filter criteria, which in turn specifies which nodes should be grayed out and which ones bolded on repaint.

5.4 Lineage Browser

The lineage browser allows users to browse the repository based on parent-child relationships, where an object is a parent of another object if the latter is derived from the former. Repository objects that are part of at least one lineage relationship are visualized as nodes in a connected, undirected graph.

5.4.1 Description

Figure 5-7 shows the lineage browser with certain nodes grayed out. Like the category browser, bolded nodes represent objects that match the specified filter criteria. Objects that match the criteria are bold, while others are grayed out. In the example given, there is a single filter restricting results to those with a name prefix of 'calculate'.

51
Clicking on a node brings that object into focus, which centers the node and lays out the remaining nodes in a radial layout, such that increasing levels of depth are at increasing distance from the root node (the node in focus). The Prefuse implementation of the radial layout is based on [23]. Double-clicking on an object sets that object to be the current Morpheus object.

5.4.2 Implementation Details

The browsing screen for the lineage browser, LineageBrowsingScreen, initializes the corresponding data browser, ItemDataBrowser, with a Prefuse data structure much like with the domain browser. This similarly allows the ItemDataBrowser to be used with other browser screens with minimal effort.

The ItemDataBrowser visualization code is based on the Prefuse RadialGraphView demo code, with the following modifications:

- Changing the backing data structure to correspond to the graph passed in during initialization
• Changing the node and label coloring actions to gray out and bold nodes based on matches to the filter criteria of the current query

• Replacing the right-click zoom controls with a mouse wheel zoom control

• Adding an item selection action to set the current Morpheus object based on the node in focus

• Adding a double-click control to trigger the item selection action

On notification of a query update, the lineage data browser requests matching repository objects as in the textual data browser, and performs node matching for coloring purposes based on object name.
Chapter 6

Related Work

Visual navigation of multidimensional spaces is a substantially active subfield of human-computer interaction. This chapter focuses on research on general multidimensional browsing systems and hierarchical visualization, two distinct but overlapping fields that are most relevant to the Morpheus browsing framework.

6.1 Multidimensional Browsing

Much of the literature on navigation of multidimensional spaces considers dimensions which are generally not hierarchical in nature. In [2], the attributes used for the purposes of navigation through a film database are “flat” characteristics such as title and director, which are not intuitively hierarchical. In contrast, genre, which is hierarchical, is only used for color coding purposes and only at a single level. Similarly, in [14] the dimensions used to browse various retail products are non-hierarchical, such as price, manufacturer, etc.

More closely tied to the goal of the Morpheus browser are systems such as Tioga, where switching between dimensions (labeled tunneling by the authors) is smoothly integrated with a zooming component that displays data at different resolutions [3]. The authors of Tioga follow the principle of Constant Information Density and ensure that at each resolution, the granularity of data changes appropriately to present roughly the same amount of data to the user [22].
6.1.1 Visual Queries

One of the key features of the Tioga system, along with the multi-level scaling and zooming, is the ability to let users dynamically create queries and result sets by connecting boxes and arrows representing different query predicates [3]. This idea of having a visual representation of queries allows users less who are less familiar with formal database query languages such as SQL to graphically construct the query to find their desired object(s), and has also been explored by various researchers [7, 8].

Some recent examples of graphical query systems include the Polaris system, where users can drag and drop different dimensions onto “shelves” in the interface, which are then used to display results using a grid system with hieraxes [19]. Another example, applied to the problem of finding real estate in Japan, is the FindFlow system presents a toolbox of various parameters that the user can drag and drop onto the main display, which contains a flow-graph display with result sets as nodes and predicates as edges [12]. Similar to Morpheus’s named query concept, FindFlow allows users to save result sets, called “packages” by the authors.

The FindFlow display system is well-suited for multi-dimensional search with primarily linear dimensions such as cost, location, etc. However, like some of the systems cited earlier ([14, 2]), it does not address the representation of specific dimensions, in particular of hierarchical dimensions, which are less easily captured by a simple predicate expression than by some kind of graphical display, whether it be a tree or other visualization. The following section addresses these issues, focusing specifically on the problem of representing hierarchical dimensions in an intuitively browseable way.

6.2 Hierarchical Visualization

Since the primary browsing dimension in Morpheus is domain, a hierarchical dimension, we also focus on research on the best way to present hierarchical data to the user. In literature related to visualization of hierarchical data, there are two concepts that are repeatedly emphasized: context and zooming. Both are related to the premise
that for any hierarchies deeper than a few levels, the sheer volume of hierarchical
information for any given focus object is too great to be easily visualized.

6.2.1 Context

The idea of keeping context displayed (often referred to as focus+context in the liter-
ature) is to have some sort of visual indication of where in the bigger picture, i.e. the
overall information space, a user is at all times. For multiscale documents, which can
be thought of as containing a hierarchy of scale, the focus+context display paradigm
is substantially faster than alternative visualization techniques [4]. The general ap-
proach for the focus+context technique is to display the current focus larger in size
and/or resolution, and display the surrounding context distorted, in lower resolution,
and/or minimized [5, 18]. The context minimization is taken to an extreme in [24],
where the context is reduced to oriented, uniformly-sized lines along window borders
and points in window corners.

One of the limitations of 2D visualization is the need to resort to distortion when
using space-filling visualizations of hierarchical data, especially for large datasets
with high branching factors. Using 3D visualization techniques instead reduces or
eliminates the need for distortion. One such technique is the Perspective Wall [15],
which presents the focus as the front face of a virtual 3-dimensional wall and the
context as angled side walls adjacent to the front wall. This technique is more suited
for linear data; hierarchical data would better fit with a visualization technique such
as Cone Trees [17].

6.2.2 Zooming

The concept of zooming is closely tied to the idea of a focus: in order to select a
new focus, especially a focus that is further down the hierarchy, the mechanism used
is often some sort of zooming technique. The focus+context techniques described in
the above paragraph usually contain an aspect of zooming, but there are also systems
which focus less on the context and concentrate on providing an informational, space-
filling overview visualization at all levels of a hierarchy [21, 18].
Chapter 7

Conclusions and Future Work

This thesis provides the basic framework for a next-generation browser for the Morpheus application, as well as a few rudimentary browser implementations. The main design goal for the framework was to create a flexible, easily extensible system that makes use of modularity and well-defined APIs to reduce code duplication, now and in the future. This was accomplished with the separation of visualization code, which is contained within the scope of data browsers, from the rest of the GUI controllers, which are handled in the higher-level browser screens. Further, the creation of the abstract DataBrowser and BrowserScreen classes allows for future development of additional data browsers and browser screens by overriding a few select methods, eliminating the need to understand every single detail of interaction between the assorted GUI components.

However, as with any ongoing project, there are several areas of improvement and additional features that can and should be incorporated into future versions of the Morpheus browsing framework. These improvements and suggested additions are described below, grouped by functional area.

7.1 Core Framework Improvements

One of the largest areas of potential improvement to the core browsing framework is in optimization, especially when fetching matching objects from the repository
(the `getObjects` methods in `NamedQuery`). Basic profiling and debugging shows that the largest bottleneck in this area comes from the JDBC call to the database; there are likely several small improvements that can be made to speed up these queries, including asynchronous data fetches, pre-fetching and storing commonly requested data, and providing the option to use a local database. Further speed gains may also be available by optimizing factors outside the scope of the browsing framework, such as providing load-balanced multiple database servers, a faster network, etc.

Refining the algorithm used to determine the ranking of objects during a data fetch would also aid users, as it would lead to a better match between the specified filters and the desired results. Currently, this algorithm is very simplistic, using only the provided Lucene scoring data and information about the intersection of all database and Lucene matches to prioritize results. Ideally, a more sophisticated algorithm would take into account factors such as number of filters matched, any weight given to particular filters, keyword distance for multi-word keyword queries, etc. Search algorithms are certainly an open area of research today, and although the visualizations provided by the browsers ameliorates the need to have a perfect search engine in Morpheus, the overall ranking algorithm should continue to improve by integrating new developments in search algorithms as appropriate.

Finally, there are a few features which were not fully implemented due to time constraints. These include the ability to add multiple filters per parameter (e.g. a maximum and a minimum cost), selecting an initial focus in the lineage browser based on the current Morpheus object, and adding date filter controls.

### 7.2 Browser Implementations

There are several areas in which the browser implementations can improve drastically. First, there are a handful of minor visualization-related bugs that need to be fixed to provide an optimal user experience, from fixing repainting of the data browsers to improving node placement in the lineage browser. There are also some relatively straightforward visualization changes that can be made to the existing browser im-
implementations to improve usability. These include easier-to-read graph layouts and using additional visual properties such as shape and size to express differences in various dimensions (size, cost, etc.).

Another extension that relates more to improving interaction than visualization would be to build a browser that links the interactions of existing browsers. For example, the current lineage and domain browsers could be combined into a single browser that "zooms out" from the lineage browser (at the item level) to the domain browser (at the category level), and automatically selects the correct domain based on what object was in focus. This type of browser would allow users to get both fine-grained information about a particular object as well as the bigger picture in the category browsing, effectively providing both focus and context as suggested in [4].

Perhaps the most interesting part of this framework is its ability to adapt to any future implementations of browsers. Even with the above-suggested improvements, the example implementations are relatively simple in their range of user interaction and visual sophistication; one can imagine far more creative implementations which may involve, for example, 3D visualizations of n-dimensional space, utilizing recent research in this area [6, 16].

7.3 User Tests and UI Improvements

After some key improvements are made to the framework core and browsing implementations, there is also some general UI work to be done. Currently, for example, only two filters may be added at a time. This was a deliberate design choice as a trade-off with a potentially cluttered filter panel, but there are likely alternate UI layouts that would accommodate the ability to add several filters at once while maintaining a clean interface. The overall look-and-feel of the browsing screens should also be modified to be consistent with the rest of the Morpheus application to provide internal consistency.

Finally, there should be usability studies conducted on both the overall interface, especially the filter and query panels, as well as side-by-side comparisons of the indi-
vidual browser screens to gauge which visualizations are most conducive for different modes of searching and browsing. This information can then be used for future design iterations of the common UI components and development of future browsing screens.
Appendix A

NamedQuery API

The NamedQuery class is the heart of the browsing framework, providing fields and methods that are used in nearly part of the framework. The nested types and selected key methods are described here in fuller detail to give a more comprehensive description of the implementation.

A.1 NestedEnumerations

A.1.1 Parameter

The Parameter enumeration represents the different metadata dimensions that can be used in a Predicate. Each Parameter is defined with a name, two string fields storing Lucene and database field names corresponding to the parameter, and two boolean flags to specify if the parameter is keyword-searchable and user-selectable. Currently, the following Parameters are defined:

A.1.2 Type

The Type enumeration represents the different relations possible between a predicate’s Parameter and value. The Type determines how a search is conducted and determines how predicates are stored.

Currently, the following Types are defined:
Table A.1: The Parameter enumeration values. Note that the SQL parameter is not directly user-selectable in the Basic tab of the filter panel, but rather has a separate textarea control in the SQL tab.

- **EQUAL**: the most basic type of predicate; this searches for exact matches

- **LIKE**: a type that allows wildcard searches in text strings; this only applies to text-based parameters such as Name, Author, and Description

- **LESS THAN**: a type that restricts the search to objects that have the corresponding parameter set to “less than” the specified value. This only applies to numeric or date/time-based parameters (where “less than” corresponds to before).

- **GREATER THAN**: Like less than, this only applies to numeric or date/time-based parameters

- **UNION**: a special type produced during a union operation on two named queries (see Section 3.3.3 for more details)

- **INTERSECTION**: similarly, a special type produced during an intersection operation on two queries

- **SQL**: a type that allows for arbitrary SQL expressions in the value; this is the only Type that does not require a specific Parameter, as SQL predicates can apply to multiple dimensions
A.2 Nested Class

A.2.1 Predicate

The Predicate nested class is a utility class representing a single predicate, a data structure that represents the conceptual filter. The class has fields for the Parameter, Type, and value associated with the predicate, and getters for each of these fields. As a convenience, hashCode() and equals() are also overwritten so that two Predicates with the same values for all three fields are considered equal.

A.3 Key Methods

The following selected methods are the most frequently used of the methods inNamedQuery throughout the browsing framework. They are grouped by general function.

A.3.1 Constructors

- public NamedQuery(String name): creates a new named query with the specified name and no predicates

- public NamedQuery(String name, Collection<Predicate> predicates): creates a new named query with the specified name and predicates

A.3.2 Query modifications

Most of these methods are synchronized to prevent concurrent modification, which could lead to an inconsistent state. Unless otherwise specified, all of these methods fire a query update notification to all registered listeners.

- public void setName(String name): set the name of the query. This method does not trigger a full update, as no predicates have changed.
• public synchronized void addPredicate(Parameter param, Type type, String value): Create a new predicate with the specified values and add it to the query

• public synchronized void addAllPredicates(Collection<Predicate> c): add all of the specified predicates, and fire a single update notification to listeners at the end

• public synchronized void removePredicate(Parameter param): remove the specified predicate from the query (predicates are uniquely specified by their parameter in the current implementation)

• public synchronized void removeAllPredicates(Collection<Parameter> c): remove all of the specified predicates, and fire a single update notification to listeners at the end

A.3.3 Data retrieval

• public String getName(): returns the name of the query

• public synchronized Predicate getPredicate(Parameter param): returns the predicate associated with the specified parameter, or null if no such predicate exists

• public synchronized Collection<Predicate> getPredicates(): returns an unmodifiable Collection view of predicates

• public synchronized boolean containsPredicate(Parameter p: returns true if a predicate exists associated with the specified parameter

• public synchronized Iterator<MorpheusObject> getObjects(Parameter... metadata: returns an Iterator over all repository objects that match the predicates associated with the query. If the optional metadata parameter is set, the specified Parameter information is retrieved and stored with the returned results
A.3.4 Listener methods

- public void addNamedQueryListener(NamedQueryListener 1): register the specified listener with the query

- public void removeNamedQueryListener(NamedQueryListener 1): unregister the specified listener

- public void removeAllQueryListeners(): unregister all listeners

- public NamedQueryListener[] getListeners(): returns all currently registered listeners


