Department of Computer Science

Jonas Lindelöw, Richard Löfberg

An analysis of the DOI framework

Degree Project (10p)
Bachelor of Engineering in Computer Science

Date:	06-01-19
Supervisor:	Stefan Alfredsson
Examiner:	Stefan Lindskog
Serial Number:	C2006:08

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

An analysis of the

DOI framework

Jonas Lindelöw, Richard Löfberg

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Jonas Lindelöw, Richard Löfberg

Approved, 19 jan 2006

Advisor: Stefan Alfredsson

Examiner: Stefan Lindskog

Abstract

This report describes and evaluates an application development framework called DOI, which is used for building document-oriented applications. A document-oriented application is an application where the user interface promotes a workflow that is tightly coupled to the business objects, instead of for example the functionality (as is the case in function-oriented applications).

The report provides a brief description of the underlying technologies, Java and Enterprise JavaBeans, followed by a description of the DOI framework. This entails technical descriptions as well as a more general overview of the framework and associated tools. There is also a more detailed description of how to create an application provided in an appendix.

In the evaluation phase we consider the following attributes: usability, extensibility, code quality and developing time. After taking into consideration how the different parts of the framework affect these attributes we conclude that DOI is a usable and extensible framework, which should contribute a great deal in the development process for applications belonging to the domain of applications targeted by the framework.

Contents

List of Figures

1 Introduction

Writing software systems to be used within a company or to be sold on the market is more often than not a huge undertaking. In some cases it might be reasonable to create the entire system from scratch. This could be the case if for example the system is of a highly specialized nature. In most cases however the development process will benefit from taking advantage of already existing functionality that is common to many applications. One area where this is important is within consulting companies that create and sell systems to be used by other companies or institutions. These companies often choose to create their own underlying frameworks that are used when building new systems. This could result both in a major decrease in development time and higher quality systems, since this framework can be written once and then be used in many different systems. When creating these frameworks it is important that they provide as much functionality as possible, while still being general enough to be useful in future systems with unknown requirements.

1.1 Background

The software development company Know IT in Karlstad has developed an application development framework in Java [1] called DOI (Document Oriented Interface). The library is intended to allow for rapid development of so called document oriented applications and is to be used internally by Know IT to create business applications. Know IT has expressed an interest in having someone develop an application with DOI in order to evaluate it. Points of interest could be for example the decrease in development time when using the library, the quality and usability of the created applications, and which types of applications that does/does not lend them self well to development using DOI.

1.2 Purpose

The purpose of this report is to evaluate the class library DOI with regards to usability, code quality, extensibility and development time. However, since the authors of this report had never used DOI before, a substantial amount of work has been put into learning how to use DOI for developing applications. In order to be able to provide a fair review of the library one needs to be in a position where enough is known to be able to use the library as intended,

in its entirety. This process of learning DOI has also been documented in the report to the extent that it provides an overview of the different parts of the library and how they work together. The knowledge gained at this stage together with the collected experiences from writing an application using DOI is later used to evaluate the class library, which is the main purpose of the report.

1.3 Disposition

The report is divided into three main parts. The first part of the report provides an overview of the underlying techniques that all DOI applications are based upon. This is the programming language Java and the EJB (Enterprise JavaBeans) [2] standard. A basic knowledge in these areas is needed in order to understand how a DOI application works, so a brief summary is provided in the report.

In the second part of the report the class library itself is described. A common approach for document oriented applications is discussed, followed by the approach taken by the DOI framework. This introduction is followed by an example of a DOI application. This example is intended to describe the "look-and-feel" of a DOI application from a user standpoint. The DOI library ships with a code-generating tool called DOI Studio, which is itself a DOI application. It will serve as the example application since it provides functionality that is common to most DOI applications. After this general introduction to DOI applications the report delves into the specific functionality of DOI Studio and describes its involvement in developing DOI applications. The final part of describing the DOI framework is a technical description, where the structure of a DOI application is described. This entails overviews of the most important classes and their internal relationships.

The third part of the report provides the analysis of the DOI class library. The library is analyzed with regards to the following attributes: usability, code quality, extensibility and development time.

Finally there is an appendix in which we show how to create a working, albeit very simple, DOI application. This application can help the interested reader to gain a better understanding of how the different parts of the framework contribute in the development process.

2 Technical overview

2.1 Java for business application development

Java as a language was initially targeted for use in embedded systems. Its first appearance in the field of personal computers was when it gained widespread use in writing plugins for web browsers, where it was intended as a plugin for small, platform independent applications that would provide more responsive web pages. In this regard Java still has its uses but it is competing with new technologies such as Macromedias Flash [3]. Since then however Java has transformed into a fully-fledged language used for writing desktop applications, and maybe the biggest use has been in business application development. The Java Platform has many features that business applications require; platform independence, the absence of complex concepts such as memory pointers, and implicit memory management through embedded garbage collection. The downside to this is somewhat decreased performance. These features make Java a language that fits the business application area, where the need for rapid development and cheap maintenance often outweighs the performance needs.

As Java began to attract attention in the business application area, developers were faced with the problem of providing services commonly used in large scale applications such as messaging, transactions, persistence and concurrency control. This led to the emergence of a new market supplying frameworks to solve these issues. As these frameworks became more common incompatibility became an issue. Applications created for one framework would not work together with a framework from another vendor. Sun Microsystems, the creator and owner of Java, realized that this was a problem and presented a solution they call Enterprise JavaBeans (EJB) [2].

2.2 Enterprise JavaBeans

Many of today’s enterprise solutions are large distributed systems. Developing these systems is a time-consuming and complex task that requires expertise in a number of areas. There are a number of services, such as persistence, security, transactions, load balancing and threading, that are needed by most enterprise solutions. All of these services that are not part of the business logic but are still needed are commonly referred to as middleware. Implementing these middleware services is time-consuming and error prone. This, together

with the fact that the middleware services needed by different applications are mostly the same, led to the development of the application server. The application server provides a runtime environment where the developer can deploy components specific to their business demands and have middleware provided for these components. This greatly reduces development time and maintenance costs. As more application servers entered the market a new issue arose. Since there were no agreed-upon standard for how components were to interact with the application servers, portability was greatly reduced because components developed with one specific application server in mind wouldn't necessarily work in another. This is where Sun Microsystems introduces Enterprise JavaBeans. EJB is a standard for developing server-side components using Java. It provides standardized interfaces for all components so that all components written to conform to the standard can be deployed in any EJB compatible application server.

The middleware services provided by the application server can be used explicitly by coding to a specific middleware API, such as performing a security check or starting a transaction. EJB however takes one step further by allowing the developer to specify the needed middleware services declaratively in an XML file separated from the source code. This information is then used by the application server at deployment time to be able to provide the components with the specified middleware. This has many benefits over the former approach, such as making the business related code easier to read. It also allows for dynamically altering the middleware services provided without recompiling any code. The following sections will provide an overview of the most important parts of the EJB specification, including how some of these middleware services are provided to the developer. For more comprehensive information about the EJB Standard and about how to use it in development, see Roman [4].

2.2.1 The Enterprise Bean

A fully-fledged EJB application consists of a number of server-side components in combination with a client that uses the services provided by these components. These components are commonly referred to as enterprise beans. An enterprise bean is made up of a number of classes and interfaces, conforming to a standard, which can be deployed to an application server. These components can range from very simple to very complex, and there is no limit on the number of java classes that can comprise an enterprise bean. The bean, however, is restricted to supply one single exposed interface. It is through this interface that

the services are made available to clients. It is not quite as simple as that though. When the bean is deployed, a few important steps are taken by the application server. First it creates an object called the EJB Object, which is a proxy for the actual bean class written by the developer. This class intercepts all calls made by the client and delegates to the bean class. This indirection makes it possible for the application server to provide all needed middleware services to the application. In addition to the EJB Object a second object called the Home Object is created. This object is responsible for creating instances of enterprise beans and providing them to the client. This gives the application server full control over all created instances, thus making it possible to provide services such as instance pooling at virtually no expense to the developer.

After seeing what an enterprise bean is made up of, the next section will examine the different types of beans available. The EJB standard defines three different types of beans: the session bean, the entity bean and the message-driven bean.

2.2.1.1 Session beans

The purpose of the session bean is to perform tasks needed by the client, such as completing a fund transfer in a banking application. The session bean will typically perform these tasks by delegation to other parts of the system, such as accessing a database or using other enterprise beans. When the client need to gain access to the services provided by the session bean it acquires an instance of the bean from the application server and calls the appropriate methods. When the client is no longer in need of the bean the application server is free to remove the instance or keep it in its instance pool for future use by other clients. The name “session bean” can be a bit misleading since a session bean does not necessarily uphold a session in the sense that it keeps state information. It could equally well be the case that a single instance of a session bean provides a service to multiple clients, without keeping any state information pertaining to specific clients. Thus one way to describe session beans is as reusable and relatively short-lived components. These components perform some specific business process, optionally upholding a session with a specific client.

2.2.1.2 Entity beans

Entity beans are persistent objects used to represent the data in the business model. As opposed to session beans the entity beans live as long as the application is in use, and they also survive application server crashes. Typically each instance of an entity bean represents

one row in a database. This is one of the key features in the EJB standard, since it provides developers with a means to model their database data as entity objects, thus cleanly separating the (typically relational) database model from the object-oriented model of an EJB application. The EJB standard also optionally provides what is known as container managed persistence (CMP). When CMP is used all storing/retrieving of data to/from the database is managed automatically by the application server, thus eliminating the need for interleaving the business code with database access code. This is a major step taken by the EJB standard to separate the database logic from the rest of the application. With CMP the application code needs no database related code, and the developers can completely envision the application data through entity objects.

2.2.1.3 Message driven beans

The third type of bean, the message-driven bean, is in many ways similar to the session bean. Similarly to the session bean it performs a specific task for the client. The difference lies in the way that the client communicates with the bean. Instead of explicitly calling a method on the bean, the client acquires the services provided by the message-driven bean by sending it messages. This makes it possible for the application to achieve asynchronous communication with the beans.

2.2.2 Transactions

Since the EJB standard adopts the client/server model one issue that will arise when developing enterprise solutions is how to handle concurrent modifications by different clients. The most common approach when trying to solve this problem is transaction handling, which means that all communication with the database will occur within a transaction. The idea is that either all operations in the transaction will occur or none will. One way to achieve this transaction handling is to manually code the transactions with the help of a transaction handling API such as Java Transaction API (JTA) [5]. This however has many of the same drawbacks as manually coding the database access. It makes the code associated with business logic harder to read and comprehend while at the same time reinstating the coupling between the database model and the object model that was avoided by using EJB container-managed persistence.

The optimal solution in most cases would be to delegate the transactions to the application server as well, similarly to how the database logic is handled, which is exactly what the EJB

standard allows the developer to do with container-managed transactions. The transaction handling needed is then specified in a deployment descriptor instead of in code. This provides many benefits, such as separating the transaction handling from the business logic, and the ability to change the transaction handling without recompiling the code.

2.2.3 Security

When developing enterprise applications a big part of the development time is often devoted to security issues, for a number of reasons. Security is a huge subject with many different techniques and with new techniques appearing frequently to handle new threats. It is also a complex subject, which substantially adds to the development time. Finally, since most enterprise applications deal with sensitive information in some way, security is an important part of the application.

Security in EJB can be provided declaratively or programmatically. EJB allows declarative security through declarations of security roles and declarations of method permissions. Thus security privileges are constrained to the method level. In some cases this could lead to problems. For example there could be cases where instance-level control is needed, as a user might be allowed to execute a specific method on only a subset of the instances of that object. In this case manual security checks would have to be performed programmatically, for example by writing code to access functionality through a security API.

3 Document Oriented Interface

Business applications in general can be categorized into different types, where applications of the same type share the same overall structure. Function-oriented and document-oriented applications are two examples of the different types of applications that exist. The traditional approach to business application development is the function-oriented model, where the workflow is guided by the functionality of the application. In these applications a window within the application represents some functionality handled by the application, such as making a transaction in a banking system. The document-oriented approach, which is the one adopted by the DOI framework and thus the one we will discuss, differs from the function-oriented approach in the sense that instead of the workflow being guided by functionality it is guided by the different business objects in the application. These business objects are then represented by what we refer to as documents. Such a document oriented workflow might for example occur in a journal handling system at a hospital where the document in focus is the journal. Another example could be the banking system mentioned earlier. In this case there would typically be a document representing a bank account, and this document would have actions associated with it to perform a transaction from that account.

3.1 The DOI approach

The DOI framework is used to create applications that model the document-oriented workflow. These document-oriented systems often share a similar architecture where the focal point is the document which contains a set of properties that can be, depending on the security credentials of the user, viewed or changed. The documents themselves are organized in different structures, and can be moved to different places or removed entirely. Searching is also something that occur in almost all such applications; displaying lists of matches to a specified search criteria. And finally report generation, since there is almost always a need for making paper copies of the documents. All of this functionality has been integrated into the DOI framework. This makes it easy for developers to simply customize the way that these services are to be provided to the end user within their application.

3.2 Architectural overview of DOI

The DOI framework is modeled on top of EJB and is thus a three-tier architecture. This is depicted in Figure 3.1. The documents in DOI are represented by EJB entity beans known as “primary entities" which can have other entity beans as children. This means that all business objects (documents) that should be represented in an application are accessed and persisted through the underlying EJB layer. The framework contains a number of classes with the purpose of both handling this representation of documents and at the same time provide the means to represent them in the application. These different classes that make up DOI can be categorized into server-side components and client-side components.

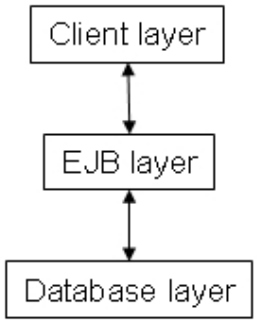

Figure 3.1: Three-tier architecture of DOI

3.2.1 Server side components

The client needs to communicate with the application server, reading and writing entity beans. Such operations could entail a number of smaller operations like EJB logic to gain access to the entity beans needed, security checks and data verification. If this entire operation is performed in the client application it will result in a number of method calls over the network, which incurs greater complexity and decreases overall performance. The solution to this problem in DOI is to use a well known EJB design pattern known as a Session Facade [6]. This pattern provides the client with an extra layer of abstraction through a session bean. In DOI this type of bean is called a broker. There is one broker for each primary entity, and it provides a common interface for the entity beans handled by the specific broker. This is a major advantage because it reduces complexity and the network traffic is concentrated to one single call since all EJB operations involved can now be performed in the application server by the broker bean.

3.2.2 Client side components

The client side components consists both of logical and GUI (Graphical User Interface) components. On the client side the documents are arranged in binders where one binder can contain one or several documents. Documents are represented graphically by two different views: an object view and a selection view. An object view has one or more panels that display an editable form of the document. A selection view displays a list of documents. The components are designed so that when a document is modified the changes are automatically marshaled to and from the server side broker. A few examples of the typical appearance of a DOI application are given by figures 3.3 - 3.8.

3.3 A DOI Application

The framework that DOI provides gives the applications that uses it a logical structure to build upon, but since the DOI framework contains a lot of graphical components a DOI application also has a certain look and feel. Figure 3.2 depicts the most important classes on the client side of a DOI application and how they relate to each other. All of these classes have a specific purpose in the application. They can however be customized to provide the functionality needed in each specific application.

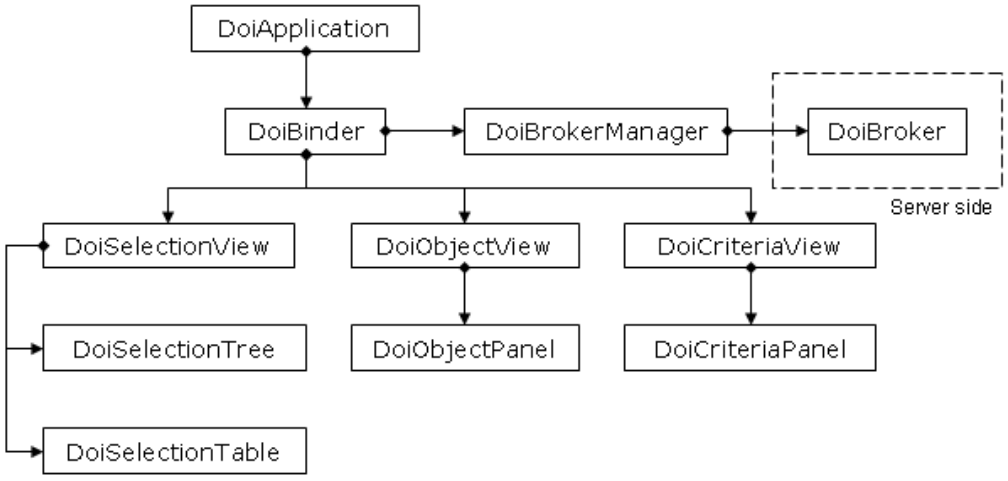

Figure 3.2: DOI class overview

To get a general apprehension of the typical appearance of a DOI application, DOI Studio will serve as an example. DOI Studio is the code generating tool that ships with the DOI framework and it is itself a DOI application.

3.3.1 Desktop and navigator

Figure 3.3 shows the main window that opens up when starting a generic DOI application. It contains three specific parts: the toolbar/menu bar, the tree view and the desktop view. The menu bar together with the toolbar with its icons is automatically generated from so called “actions”. All of the menus and icons that can be seen in the figure are default actions that come with all DOI applications, specific actions can if desired easily be added or customized. The tree view is showing the first level containing the primary entities. At deeper levels in the tree view implementation specific nodes can be added. The desktop view is a multiple document interface (MDI) view that can display multiple windows representing for example business objects. This is the type of interface that can be seen in for example older versions o Microsoft Word.

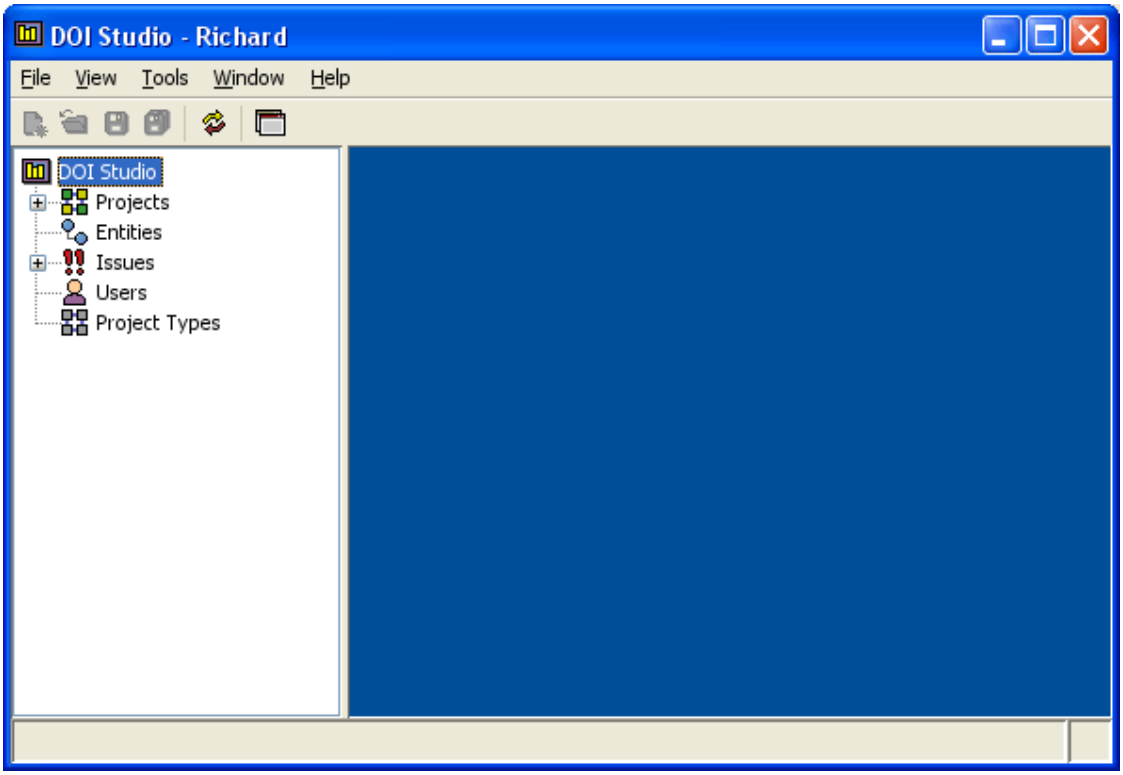

Figure 3.3: Desktop and navigator in DOI Studio

3.3.2 Selection table

In the first level of the tree view at the left in Figure 3.4 there is a list of the primary entities that the application is built upon. To show a list of all the entities of a specific type that exists one can double click on the corresponding node in the tree. In the figure we have selected the “Entities” node and are currently shown, in the desktop view, a list of all “entity”

entities beginning with the letter C or D. Most of the selection table's functionality is implemented in the DoiSelectionTable class that the specific implementation extends. Because the graphical component that represents the table itself is a specific DOI component, customizations such as what columns to show together with their format and so on is easily performed in a Java GUI editor such as NetBeans [7].

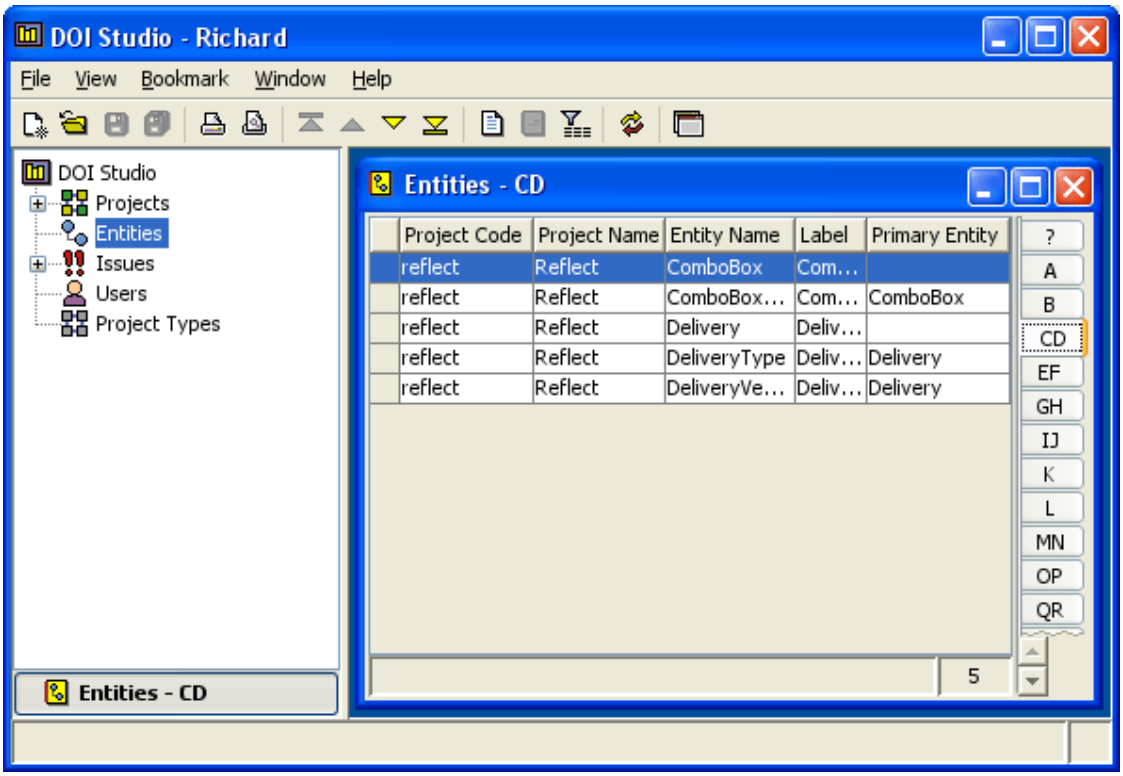

Figure 3.4: Selection table in DOI Studio

3.3.3 Selection criteria and bookmarks

The selection table can quickly become unwieldy with a growing number of entities. Here a searching and sorting function of some sort becomes sensible and this is easily supported by the DOI framework. The support comes from what is called selection criteria and bookmarks. The selection criteria are represented by a panel that extends the functionality of the class DoiCriteriaPanel. The base class provides the basic functionality together with the buttons at the bottom of the window in Figure 3.5. All the developer has to do is to, using a Java GUI editor such as NetBeans, place the text fields and name them properly. The user fills in the appropriate criteria and chooses "apply".

In Figure 3.5 it is specified that all entities with the project code “Reflect” is wanted. When applied this will result in the selection table we see in Figure 3.6 with a listing of all entities with the project code “Reflect”.

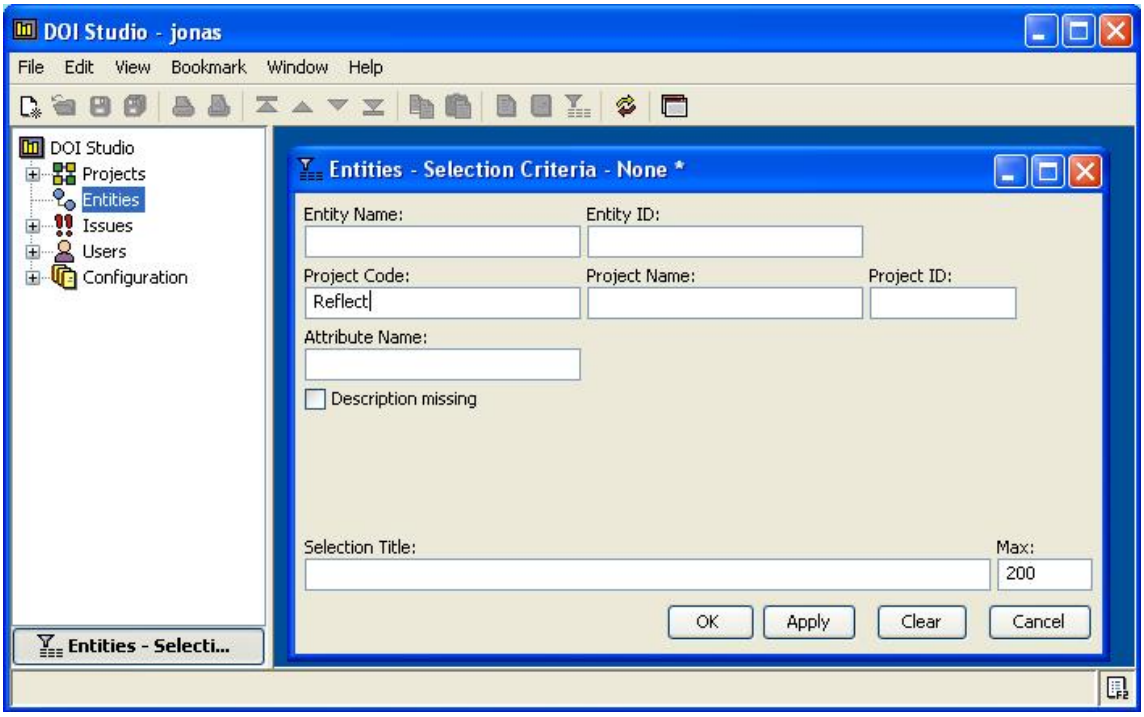

Figure 3.5: Using selection criteria in DOI Studio

Selection criteria that are used often can be saved by the user as a bookmark. In Figure 3.6 the selection criteria has been saved as “Reflect entities” and it is shown as a bookmark under the “Entities” node. When a user selects this bookmark they will again be shown the filtered selection table. Bookmarks can also be shared amongst users depending on their authorization.

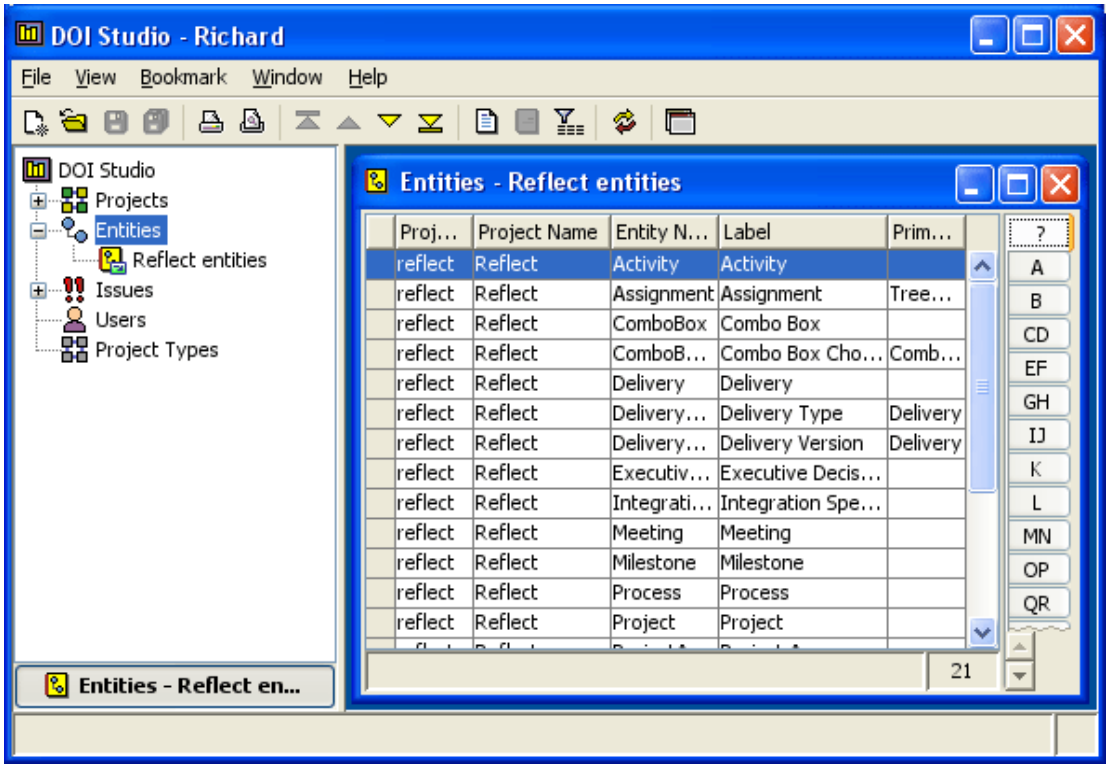

Figure 3.6: Results of applying selection criteria in DOI Studio

3.3.4 Object view

The object view is what represents the actual document in the system and is thus not easily generalized. Figure 3.7 shows the object view for the “Entity” entity in DOI Studio. The DOI classes that are extended for this purpose are DoiObjectView and DoiObjectPanel. An object view can have one or more panels. When an object view has more than one panel, each panel is displayed in a tab of its own. Figure 3.8 shows an additional panel being activated in the “Entity” object view.

The development of the object panels is usually quite straightforward. Using a Java GUI editor such as NetBeans, components are graphically added to build up the document. There are custom DOI components for general input such as textboxes, drop down boxes, tables and so on, but generic Java components can also be used. The custom DOI components remove most of the need for custom coding. The developer simply drags the component to the panel and enters its name based on a special naming scheme consisting of the entity name and the field name. The data in these components will then automatically be updated by the running system.

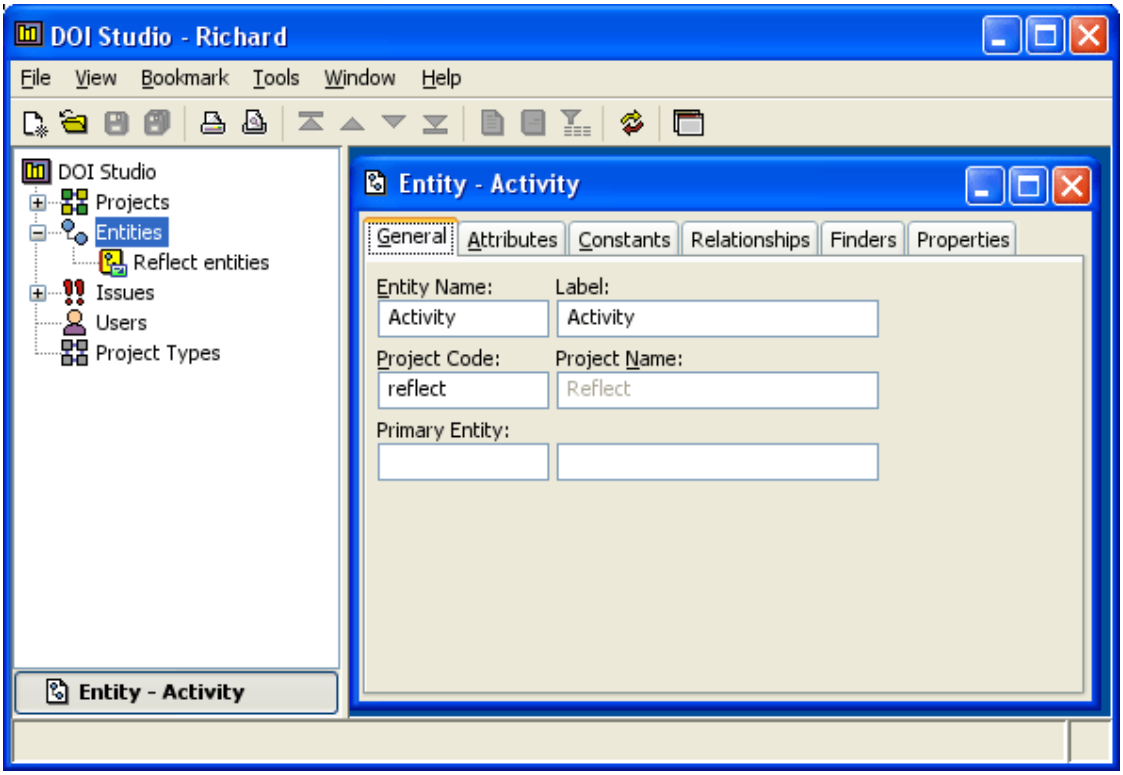

Figure 3.7: The object view in DOI Studio

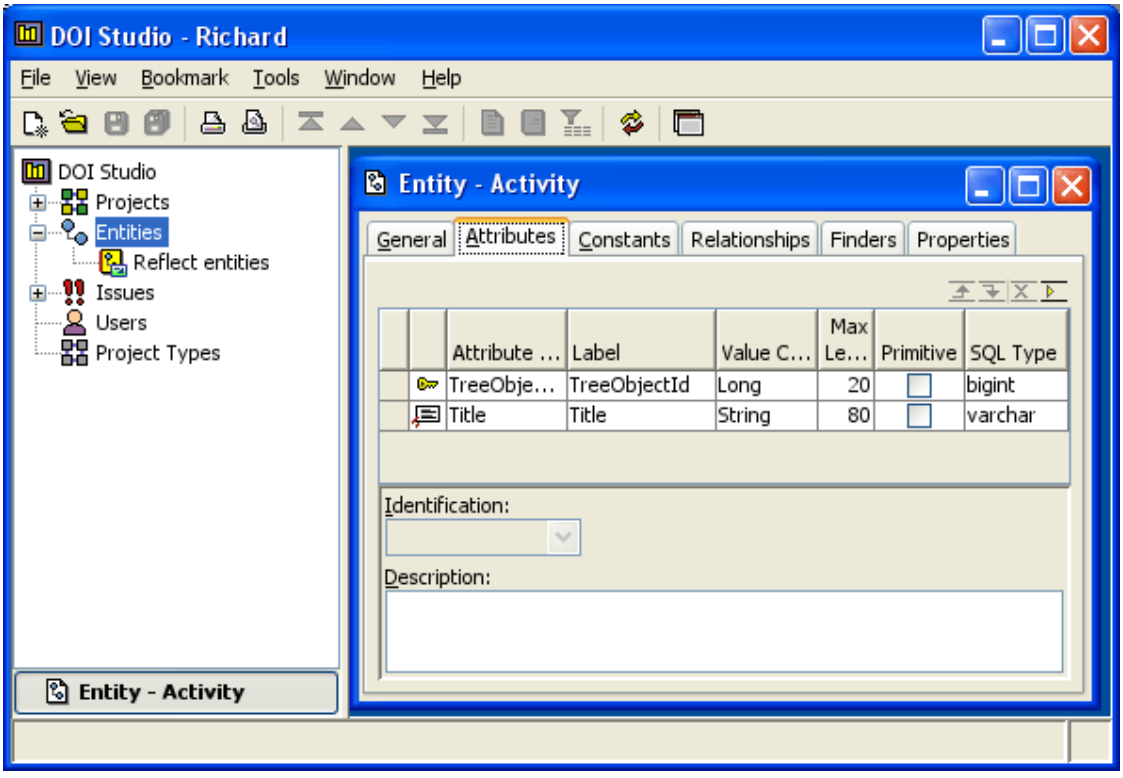

Figure 3.8: Additional panels of the object view in DOI Studio

3.4 DOI Studio

DOI Studio is a code generation tool that allows for rapid development of DOI applications that ships with the DOI framework. When using DOI Studio all entities and properties of an application are specified in a graphical user interface, and all this information is stored in a relational database. This "graphical" development can proceed until the application is correctly described in DOI Studio. When this state has been reached the program has the ability to generate a NetBeans project, complete with source code for the client application and all enterprise beans (entity beans to model entities, session beans to provide access to the entity beans).

After a project has been generated using DOI Studio, development of the application can proceed in NetBeans, where a rich set of specialized GUI components provided by DOI further speeds up the process. If circumstances demand that new entities be introduced into the project, or other properties need to be changed, then DOI Studio can once again be invoked to regenerate the source with the incorporated changes. In this case a merge tool would preferably be used to differentiate between generated changes and manually edited code. This cycle of alternating between development in DOI Studio and NetBeans can continue throughout development as needed.

3.4.1 Creating a database

When a new project is created in DOI it has to be designated its own underlying storage somewhere, to be used by the application at runtime. So the first step in generating a DOI application with DOI Studio is to model the underlying storage. This is a relational database with tables for all the entities that should be represented in the application. As an example, in a banking application an entity could be a bank account, with attributes such as the account number and the current balance. This has to be done with appropriate external tools.

3.4.2 Creating a project

With the database in place the next step in the process is to create a new project in DOI Studio. The project is given a name and a database source is pointed out through a JDBC (Java Database Connectivity) [8] name that is created beforehand through the application server’s administrative tools. There are also numerous other properties that can be set for a project in DOI Studio but those will not be covered in detail. When all this is done the final

step before generating the project source code is to provide DOI Studio with information about all entities and the relationships between these.

3.4.3 Populating a project

When all entities have been correctly modeled in the underlying storage the task of adding them in DOI Studio is very straightforward. The first step is to create a new entity and giving it a name corresponding to the table name in the database. As soon as this is done DOI Studio can automatically generate all the entity attributes from the table attributes. The entity can also be designated to have another entity as its primary entity. If no such entity is designated the entity itself will become a primary entity. This has an impact on how the application will represent the entity, with regards to, for example, which (if any) graphical windows will be created to display it to the user. When this has been done for all entities in the project DOI Studio has full knowledge about all entities and their attributes together with an imposed structure on these entities as primary or secondary entities.

The relationships between the entities in the underlying storage should also be represented. These relationships will have been modeled in the database according to the relational model, with foreign keys. These foreign keys should be named by a specific naming scheme imposed by DOI Studio. Assuming this has been done correctly DOI Studio can import the entity relationships automatically as well, as was the case with the entity attributes. When the source code is generated these relationships will be represented by methods in the entity beans, according to the EJB standard.

3.4.4 Generating source code

When all of the above steps are complete DOI Studio has all information needed to generate a complete application. This is done in two steps. In the first step all source code is generated but kept in memory, with a checksum added to the bottom of each source file. By doing this the program can inform the developer of whether the files are new, updated or manually edited, the last of which would be the case if the file had been regenerated at a later stage in the project, after being manually edited. The developer can then choose which files to write to disk. There is also an option available to run the new version of a generated file against the old version through a merge program, thus manually resolving the differences instead of allowing DOI Studio to write the file to the source directory. After this final step,

the project directory on disk contains a complete NetBeans project, ready to be compiled and deployed to an application server.

3.5 Technical description

This section will provide an overview of the different parts of a DOI application. These parts are all generated by DOI Studio and can later be extended by developers to provide a higher degree of functionality. Since DOI is a big library we will only mention the basic underlying parts that are common to all DOI applications.

3.5.1 DoiApplication

The project created by DOI Studio is a complete enterprise application, with classes generated for both server components and the client application. The client application is an ordinary Java application and the entry point of this application is a class derived from the class DoiApplication. DoiApplication provides functionality for handling most of the other parts of the application generated by DOI Studio. These DOI specific classes will be covered later. It also provides the rest of the application with such things as access to top-level windows, security credentials, login procedures, and much more. A large degree of this functionality is common to most applications and therefore will be inherited in the specific class created for an application, but if needed these methods can be overridden in the inherited class to customize the behavior of the application.

3.5.2 DoiBroker

DOI Studio generates entity beans for all entities modeled in the project. The client then needs to perform business operations involving these entities, such as for example making a deposit into a bank account. In DOI these operations are performed through brokers. One broker is generated for each primary entity in a DOI application. The methods provided by the brokers in a DOI application provides all needed functionality to create, delete, read and write entities.

On top of the basic functionality the brokers provide an extra degree of abstraction when reading or writing entity beans. For example, when reading data from an entity bean the client receives this data in a map, with the field names used as keys. This map then becomes the representation of the entity data on the client side. This model also allows for the broker to provide the client with so called parts, which is a named collection of data pertaining to some

specific entity. This collection could be for example a subset of the fields of the entity, or even a table representing some relations that this entity has to other entities. To better understand the principle, the following example will show how this is used when creating a graphical interface for an entity in a DOI application.

Suppose that there is an entity **Customer** with the fields *CustomerId* and *Name*, and an entity **BankAccount** with fields *AccountId* and *Balance*. There is a 1-M relationship between **Customer** and **BankAccount**, since one customer can have several bank accounts. When creating an interface for a customer, one might want to display the *CustomerId* and *Name* fields within one panel, while displaying the customer's bank accounts in another panel within a table. One way to do this with DOI is to create the two panels and add the components needed to display the data. The components are then given names corresponding to the fields they represent. So *CustomerId* could be represented by a text field with the name **Customer**.*CustomerId*. When this is done names are provided for the panels as well. In this example the panel with customer information is named "General" and the panel displaying accounts is named "Accounts". This is basically all that needs to be done. What will happen when the program is run is that when we choose to save the view the underlying code will collect all data from each panel into a value map with the names of the components as keys. Then, for each panel, the method writePart<PanelName>() will be invoked in the associated broker, with the value map sent as a parameter. The broker is then responsible for correctly updating the entity bean with the data provided by the user. The writePartGeneral() method and methods for parts representing all relationships an entity has will be provided by default in all brokers. If a more customized view is needed then all that needs to be done is to write new part methods in the broker. This example covered the writing of a part. For each writePartXXX() method there is also a corresponding readPartXXX() method and a validatePartXXX() method. The validate method is called before the write method to ensure that only valid data has been entered by the user.

These parts mentioned is the main method with which the brokers in DOI provide the client with access to the underlying entity beans. This approach abstracts the EJB specifics while at the same time providing a flexible solution to the problem of reading and writing data. It is also worth mentioning that in cases were multiple parts need to be read or written it is still done within a single transaction, so when a failure occurs all operations are rolled back.

3.5.3 DoiBinder

Each primary entity in a DOI application has associated with it a so called binder, which is a class sub-classed from DoiBinder. Each binder is responsible for managing all views for a specific primary entity. The binders in DOI provide a lot of functionality to the newly generated application, such as methods to print views, the ability to specify how and when different views should be opened and how the entity should be represented in the tree structure that is common to all DOI applications. Besides providing this basic functionality to all DOI applications, the fact that each binder is sub-classed makes the binder a natural location wherein all functionality pertaining to a specific entity can be implemented in the client application.

3.5.4 DoiObjectView

One of the views managed by each binder is the object view. The object view is a window used to display one specific entity instance. As an example, assume that in the banking application from earlier there was a binder associated with the entity bank account. This binder will then be responsible for managing all object views for this entity, where each object view displays information about one specific bank account.

Each object view keeps track of the primary key of its current entity instance. This simplifies the process of storing and retrieving the data displayed in the view. It is also easy to programmatically obtain the view associated with a specific entity instance, if one is currently available. These and many other useful functions are already implemented when creating a new application, either in the DOI library or in the code generated by DOI Studio. The window itself however is only a blank window, which makes sense since the manner in which to display the data varies between applications. The purpose of the object view is to provide the developer with a "blank canvas" where the graphical representation of an object can be designed, while at the same time providing a wealth of underlying functionality needed by most DOI applications.

3.5.5 DoiSelectionView

A selection view displays to the user a collection of entities of some specific type. For example there could be a bookmark in the applications navigator view that represents all instances of some specific entity. When activated a selection view would then be created to display these entities. The default implementation generated by DOI Studio displays a table

where the primary key of the entity is one of the columns. Each row in the selection table can then be activated to open an object view for that specific entity.

The appearance of the selection view for a particular entity can be customized as needed by the developer. The alterations could be as simple as adding columns to the table or as involved as removing the default implementation entirely and provide a customized interface. Generally however, what is often needed by an application is to be able to display a collection of entities and some selected information about these, together with the ability to navigate to the individual entities, which is exactly the functionality provided from DOI Studio by default.

3.5.6 DoiCriteriaView

In section 3.5.5 it was shown that one way to be provided with a collection of entities is by activating an already stored bookmark. This bookmark represents a fixed collection of entities that has either been created explicitly by the application or stored by a user. In many cases an application also needs to provide the user with a way to acquire a selection of entities based on some specific criteria. This is the purpose of the criteria view. It provides the user with a graphical user interface where search criteria can be specified for some entity. These search criteria are then used in conjunction with a selection view to display the results to the user. The user also has the option to then create a new bookmark for these search criteria. This provides for a flexible way to dynamically organize information at the user level.

4 Analysis

In the analysis of the DOI framework the following attributes was considered: usability of both the applications and the developer tools, quality of the generated code, extensibility and improvement in development time. These attributes were considered important in a library that is going to be used continuously to develop new applications.

Usability is important both from an end-user perspective and from a developer perspective. Ideally all applications written with the framework should provide the end-user with a consistent and intuitive working environment. From the developer perspective usability is affected by factors such as which tools are made available for developers and to what degree different parts in the development process are automated.

The quality of the generated code is important because this is the code that will be read and customized by the developers of applications. Keeping this code readable and within reasonable limits of complexity is important for effective development using the library.

The extensibility of the framework measures its ability to adapt to different requirements. This is important since it is difficult to foresee the requirements of future applications. Ideally the framework should provide a high degree of functionality while still being easily customized to handle unforeseen requirements.

All of the attributes mentioned so far are linked together to some degree and all of them contribute to the last attribute, which is the improvement in development time when using the framework. Decreasing the time spent developing applications is one of the overall goals of using the framework in the first place and is thus an important factor in the analysis. In the following sections we have analyzed each of these attributes and the factors that affect them. Although some coupling between the different attributes analyzed is unavoidable, each attribute has been analyzed separately.

4.1 Usability

In the case of usability it has been analyzed both from the end-user perspective and from the perspective of the developer. The end-user usability primarily affects the overall quality of developed applications, while the developer usability affects the complexity and workload in the development process, thus also affecting development time. It should be mentioned that this is a rather informal analysis. For more comprehensive information in this area, see for example the works by Jacob Nielsen [9].

4.1.1 End-user usability

Applications written with the DOI framework provides the end-user with an interface that models the document-oriented workflow. This provides the user with an intuitive view of the business objects as documents, where a document typically represents a single business object. The applications are also intuitive in the sense that they strongly resemble many other business applications on the market today, both in graphical style and basic functionality.

The behavior of the typical DOI application differs from the general formula in some aspects, and in our opinion these changes are mostly for the better. A few differences in particular come to mind. One difference is the sparse use of so called combo-boxes, the drop-down text fields that are commonly used to select between a pre-defined set of text strings. Another apparent difference is that DOI applications in general do not include Ok buttons for applying changes done by the user. The combo-boxes are, to a large degree, replaced by so-called lookup fields. These lookup-fields allow the user to enter part of the value and get a list of matches by pressing a key. Combo-boxes still exist but they are now used mainly to display choices that are completely static. The absence of Ok buttons is inherent in the fact that the windows in an application should be seen as documents. The approach taken by DOI applications is that an edited document is saved with a keyboard shortcut or by pressing a designated button in the menu. This differs from the approach taken by most function-oriented applications where more commonly a window represents some specific functionality provided by the application, where the changes are propagated by pressing an Ok button, which also closes the window. The main difference is that instead of saving a document representing a business object, in a function-oriented application we tell the application to perform some specific functionality. This functionality may very well make use of the business objects in the application, but the way in which these objects are envisioned in the application differs from the document-oriented model.

These differences might seem minor, but in many cases they have an effect on how the application is used by the end user. For example, the lookup fields, together with several other components in a DOI application, are designed to be easy to navigate and edit by only using a keyboard. This could have a major impact for the end-user whose only job might be to enter and edit information into some information system.

Another useful feature in DOI applications are bookmarks which, especially in combination with selection criteria, allows the user to customize his/her own interface to more easily gain access to relevant information. The fact that all of the functionality described makes sense in virtually all applications in the document-oriented domain makes it a valuable addition.

4.1.2 Developer usability

The usability of the framework from a developer perspective is tightly coupled with the improvement in development time, so many of the factors taken under consideration here are relevant to the later analysis of development time as well. When analyzing developer usability we considered the ease with which the framework is used by the developers.

One way in which DOI enhances the usability immensely is through DOI Studio, which provides an intuitive and fast way to get the application up and running. Not only does DOI Studio help developers with creating the Java sources. It creates an entire project, complete with EJB deployment descriptor and XML build files for automatically building the project in NetBeans. The automatic generation of all these parts is no small accomplishment, since this is a major part of every application that uses EJB as the underlying technique.

One impressing feature regarding developer usability in DOI is the clean separation of the EJB subsystem from the rest of the application. It was mentioned earlier that session beans called brokers handle communication with the underlying entity beans. This means that a developer could generate a DOI application with DOI Studio and then write the application's business logic by using the functionality provided in the brokers. This could theoretically be done without any prior knowledge of EJB. This should be a great advantage for developers in big development projects, where maybe only a few of the project members will need to learn EJB. It should also be said that in no way does DOI restrict the developer from taking

advantage of his/her knowledge about EJB when it is needed. It might for example be necessary to add or change functionality in the brokers to provide for specific functionality in an application.

The analysis of the developer usability is concluded by considering the graphical components provided by the DOI framework. These components provide functionality commonly needed in many applications, such as text fields and tables. However since these are specific DOI components they are designed to easily interact directly with the underlying storage. In many cases the only thing needed in order to make a component automatically persist itself is to give it the correct name. These components greatly simplify the task of building graphical user interfaces, which is often time-consuming and repetitive.

4.2 Quality of generated code

When assessing the quality of the generated code we used our own experiences from developing an application in DOI. We have tried to express our opinion of how easy the code is to read and change, together with the overall complexity of the code. We have also looked at the coding standards used in the generated code.

4.2.1 Readability

We have had very few problems with reading and understanding the code. All methods are extensively documented. In fact we learned to use the DOI framework almost exclusively by reading existing code. Readability is further enhanced by the fact that descriptive and logical identifiers are used throughout the code. Most times the purpose of a function or variable can be determined simply by referring to its name and type.

4.2.2 Writability

After writing a few applications using the DOI framework our opinion is that it is relatively straightforward to write new. The framework provides the developer with many code structures and layers of abstractions that makes it possible to achieve a very high degree of functionality in relatively few lines of code. We have found in our applications that most methods consists of fairly few lines of code (generally 10-20), even the more complicated ones. This implies a high degree of writability and reuseability, while at the same time improving readability.

4.2.3 Complexity

The DOI framework addresses an inherently complex area of development: distributed enterprise applications. Our opinion however is that DOI does a good job of hiding much of this complexity in the generated code. The underlying complexities are still there but in most cases the developers can work against simpler interfaces to achieve the desired functionality.

4.3 Extensibility

The purpose of the DOI framework is to create applications that follow the document-oriented workflow. This in some ways allows the framework to take one step beyond a purely general framework and make some assumptions about the functionality needed in applications written with the framework. It is important however that the framework, while providing this functionality, is still easily extensible. The DOI framework approaches this dilemma by providing a highly modularized class structure, where most of the classes in the generated code are sub-classed from some class in the library that provides the basic functionality needed by most applications.

The fact that classes are sub-classed in the final applications makes it easy for a developer to customize, remove or replace functionality inherited from the base classes in order to better suit his/her needs. As far as our collective experiences go we never felt restricted by the underlying structure of the DOI framework. As an example from one of our own applications, at one time we needed to change the way the graphical tree view worked in the user interface. We did this by writing our own classes representing the tree nodes and the tree itself, and then proceeded to replace the one provided by the DOI framework. This was done in less than a days work and had no unforeseen negative side effects at all.

4.4 Improvement in development time

During the course of writing this report we have written a few applications using DOI. From our own experiences in writing similar applications without the help of a framework we have tried to pinpoint the different aspects of the DOI framework that contributes the most in decreasing development time.

4.4.1 DOI Studio

The code generating tool in the DOI framework, DOI Studio, has a major impact on development time in that it relieves the developer from the responsibility of manually coding all EJB related classes etc. For simple applications as the ones we have been developing during the work on this report, the entire application is often generated in DOI Studio with a maximum of a few hours of work. This is a major reduction in time as compared to the time it would take to implement it by hand, not to mention that doing it by hand is error prone and could introduce low-level bugs in the application. These bugs could increase development time even further down the line. Finally there lies a lot of thought behind the code generated by DOI Studio. This results in a robust underlying system which new applications can build upon. It should also reduce the need for refactoring during later stages of projects. This is due to the fact that the entire EJB sub-system is already thoroughly tested and debugged.

4.4.2 The DOI framework

The DOI framework provides applications with a very high degree of functionality. Even more so than most general-purpose application development frameworks. This is made possible partially by the fact that the DOI framework is intended for the subset of applications within the document-oriented domain. This means that the framework can make certain assumptions about the functionality of the application and therefore impose an underlying structure in the applications. This has the unavoidable side effect that not every application lends itself well to development using the DOI framework. This said, our experience has been that most parts of development have taken much less time when using DOI than it would have otherwise, at least under the assumption that the requirements of the applications does not deviate too far from the structure imposed by DOI.

4.4.3 GUI components

The specialized GUI components provided by the DOI framework have proven themselves to be major time-savers when developing user interfaces. The common approach to developing user interfaces often requires a lot of code to connect the graphical components to the underlying system. This could be for example code that retrieves or sets values of the components and uses them either as a return value or as a parameter to some method in the underlying system. The DOI components however are already designed to automatically persist themselves using simple naming schemes. In many cases no manual editing of code is needed at all.

4.4.4 Learning curve

All the factors mentioned does of course have a negative side effect in that the developers need to learn using the framework. Our opinion is that in learning to use the DOI framework the most important aspect is to learn and understand the principles of how a DOI application is supposed to work and how the different parts of the library relates to each other. A substantial amount of time was put into learning the different parts of the library before writing this report.

It should be mentioned however that in our case the circumstances where not ideal. This is due to the fact that the framework is newly developed and the documentation was undergoing work during the time we were using it. We have thus relied mostly on reading code. This worked for us since the code is exceptionally well documented in itself, but we still predict that the learning curve will be flattened considerably as developers are provided with more comprehensive documentation, including tutorials on how to develop applications using the DOI framework. This documentation was near completion at the time this report was finished. It includes a more advanced example application than the one provided in the appendix of this report.

5 Conclusions

We have examined a development framework (The DOI Framework) used for developing applications using Java and Enterprise JavaBeans. We have used this framework for practical purposes in order to be able to analyze its contribution to the development process. During this work we have concluded that the DOI framework should significantly decrease development time. This is due to the functionality provided by the classes in the framework, in combination with the code generation tool which relieves developers of a lot of coding. The only downside is that the framework, since it targets a specific type of applications, might not fit all applications. Our opinion however is that the benefits provided by the DOI framework makes this a reasonable tradeoff.

We have also concluded that there lies a lot of thought behind the structure of the framework and the generated code. We feel that the framework does a good job of hiding away unnecessary complexity from the developer, while still providing a high degree of functionality. The code is of high quality throughout the entire framework and it is extensively documented. This makes the framework easy both to learn and to apply during development. The structure of the framework also provides a high degree of extensibility, in the sense that it is easy to customize the behavior of applications within reasonable bounds.

Overall, we have enjoyed developing applications using the DOI framework and our opinion is that it provides great benefits when developing the kind of document-oriented applications that the DOI framework targets.

6 References

[1] Gosling, J., B. Joy, G. Steele and G. Bracha. *Java(TM) Language Specification – Third Edition*, Addison-Wesley, *(ISBN: 0321246780).* Can also be read in full at http://java.sun.com/docs/books/jls/, *hämtad den 23 januari 2006.*

[2] Sun Microsystems. *Enterprise JavaBeansTM Specification*, http://java.sun.com/products/ejb/docs.html, *hämtad den 23 januari 2006.*

[3] Vogeleer, D. *Macromedia Flash Professional 8 Unleashed*, Sams, *(ISBN: 0672327619).*

[4] Roman, E., R. P. Sriganesh and G. Brose. *Mastering Enterprise JavaBeans - Third Edition*, John Wiley & Sons, Inc, *(ISBN: 0764576828).*

[5] Sun Microsystems. *Java Transaction API*, http://java.sun.com/products/jta/, *hämtad den 23 januari 2006.*

[6] Marinescu, F. *EJB Design Patterns*, John Wiley & Sons, Inc, *(ISBN: 0471208310).*

[7] Sun Microsystems. *NetBeans*, http://www.netbeans.org, *hämtad den 24 januari 2006.*

[8] Sun Microsystems. *Java Database Connectivity API*, http://java.sun.com/products/jdbc/, *hämtad den 24 januari 2006.*

[9] Nielsen, J. *Usability Engineering*, Morgan Kaufmann, *(ISBN: 0125184069).*

[10] DuBois, P. *MySQL – Third Edition*, Sams, *(ISBN: 0672326736).*

[11] *ALS Microsoft(r) SQL Server 2000 System Administration*, Microsoft Corporation, *(ISBN: 0735614261).*

A An example application

In this section an example application will be built from the ground up using DOI. The application will model a simple DVD collection with information about movies and actors.

A.1 Database design

The database design is fairly simple and is shown in Figure A.1. Each movie will have associated with it a title, a length in minutes, a production year and a short summary (max 255 characters). Each actor will in turn have associated with it a name, a date of birth and a nationality. Finally, the Cast table models an M-M relationship between the Movie entity and the Actor entity.

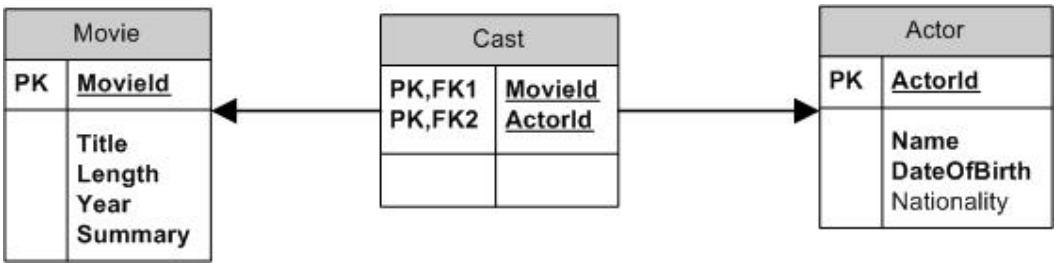

Figure A.1: Database design of the DVD collection example application

The database could be realized using either MySQL [10] or Microsoft SQL Server [11]. In this case we have chosen to use MySQL and the accompanying MySQL Query Browser, which is a graphical tool for creating and querying MySQL databases.

The specifics of how to create the database will not be covered here but there are a few issues worth mentioning. To begin with, it is generally a good idea to name all fields with capitalization according to Java coding standards. This way we do not have to manually edit these names later in DOI Studio. One other thing worth mentioning is the naming of foreign keys. When creating foreign keys in MySQL (or Microsoft SQL Server) they should be named according to the naming standard provided by DOI Studio. This naming standard mandates that foreign key names be on the form fk_RoleName1_RoleName2. For example table Cast will have a foreign key named fk_Actors_Movies and one named

fk_Movies_Actors. See Figure A.2 for an example of how to name the foreign keys in MySQL.

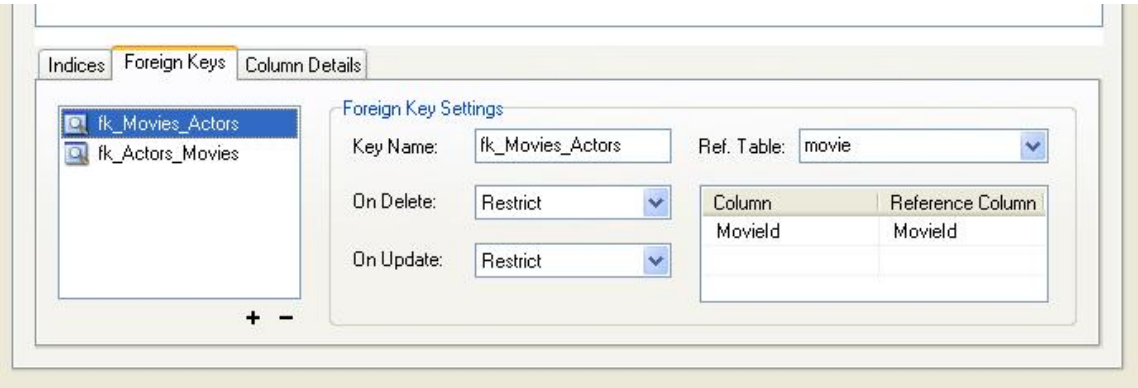

Figure A.2: Naming foreign keys in MySQL

A.2 Setting up the application server

With the database in place there are a few more steps that needs to taken before starting up DOI Studio to generate the application. This includes setting up the JDBC source in the application server in order to be able to connect to the database. We will also need to create a user with permissions to alter the newly created database and register this user in the application server. This procedure differs with the choice of application server.

For future reference, in this example we have named the JDBC source jdbc/ExampleApp. The database user is named “exampleapp” with password “exampleapp”.

A.3 Generating the project with DOI Studio

We are now ready to start generating the application in DOI Studio. This entails the steps covered in the following sections.

A.3.1 Creating a new project

After starting up DOI Studio we right-click the projects node in the tree structure and choose “New” to create a new project. We are then presented with an object view where we will enter information about the project. Figure A.3 shows the “General” panel of this view where we have already entered information. We have chosen a project code, a name for the project, and a project type. We have also chosen a root directory for our project. This is the

directory into which the application will eventually be generated. Finally we have pointed out the correct JDBC source. The options for JDBC sources are collected directly from the application server by DOI Studio.

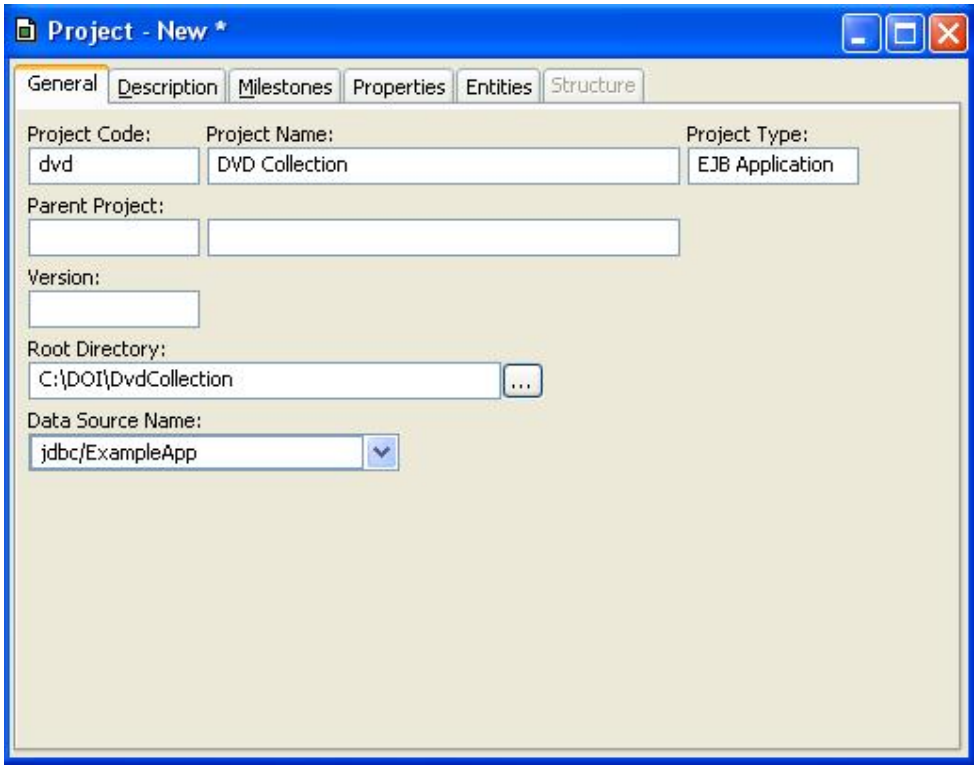

Figure A.3: General project information in DOI Studio

When the information has been entered we notice an asterisk appearing in the title bar of the window. This means that the information in the object view has been saved and that the view is eligible for saving. This can be done in two ways: either by clicking the disk icon in the toolbar or by using the keyboard shortcut CTRL-S. When the information about the project is saved DOI Studio will persist all the information to a relational database.

A.3.2 Setting project properties

At the same time the view was saved, a set of default properties was created for the project by DOI Studio. These properties are viewable under the “Properties” tab and the next task will be to edit them for our project. In Figure A.4 we have chosen values for the properties. The checkboxes in the column “Used” determines which properties that have values supplied by the user and which will have the default value. The “Base Package Name” and “Broker Manager Name” properties affects the naming of the projects Java packages and the name of the broker manager that keeps track of all brokers, respectively. The “DOI Home” property

points out the directory where the DOI library is installed. The remaining checked properties should be self-explanatory.

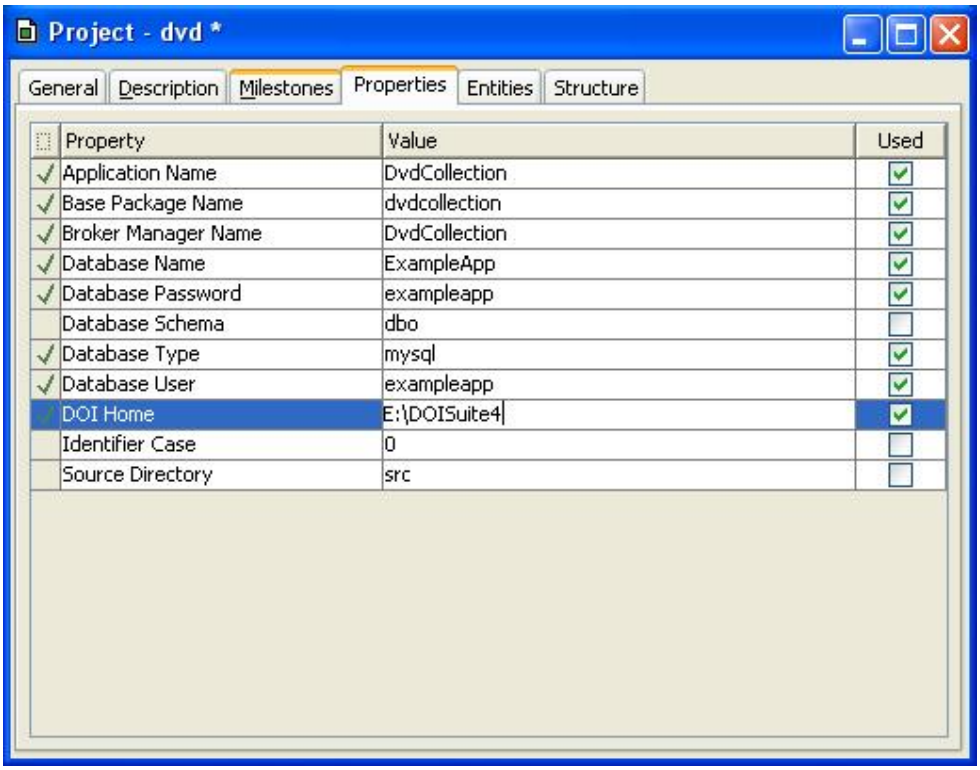

Figure A.4: Properties for a project in DOI Studio

A.3.3 Creating entities

Since we have now created a project in DOI Studio we could use the built-in code generator to generate a complete NetBeans project. But before doing that we will provide DOI Studio with information about the entities we wish to model. In this case we want to model the entities Movie and Actor, together with the relationship between these.

We begin creating the entity Movie by right-clicking the Entities node in the tree and choosing “New”. What we now see is an object view representing a new entity. In Figure A.5 the “General” tab of this object view is shown with information entered into the relevant fields. The entity name will be the name of the entity in the code generated by DOI Studio and this name should correspond to a table name in the underlying storage. Thus in this case we have named the entity Movie. We have also specified to which project this entity belongs by entering the project code chosen earlier. The project name will then be automatically provided by the application. The primary entity should be provided in the case when the entity

is a secondary entity. In our simple example both Movie and Actor will be modeled as primary entities so this field will not be used.

At this stage the information about the entity should be saved to storage before we proceed, since DOI Studio will use information from the database in the following steps.

Figure A.5: Information about an entity in DOI Studio

A.3.4 Importing attributes

We will now add the attributes to the newly created entity from the database. This process is automated by DOI Studio. When viewing the "Attributes" tab of the entity object view, the attributes are imported by choosing "Tools/Import Table Columns" from the menu bar. In some cases some of the information such as data types or attribute names needs to be manually edited at this stage. In our case we choose to save the information as is. In Figure A.6 the attributes tab is shown with attributes imported for the entity Movie. As we have now completed the Movie entity we repeat the same process for the entity Actor. This will not be shown in this appendix however.

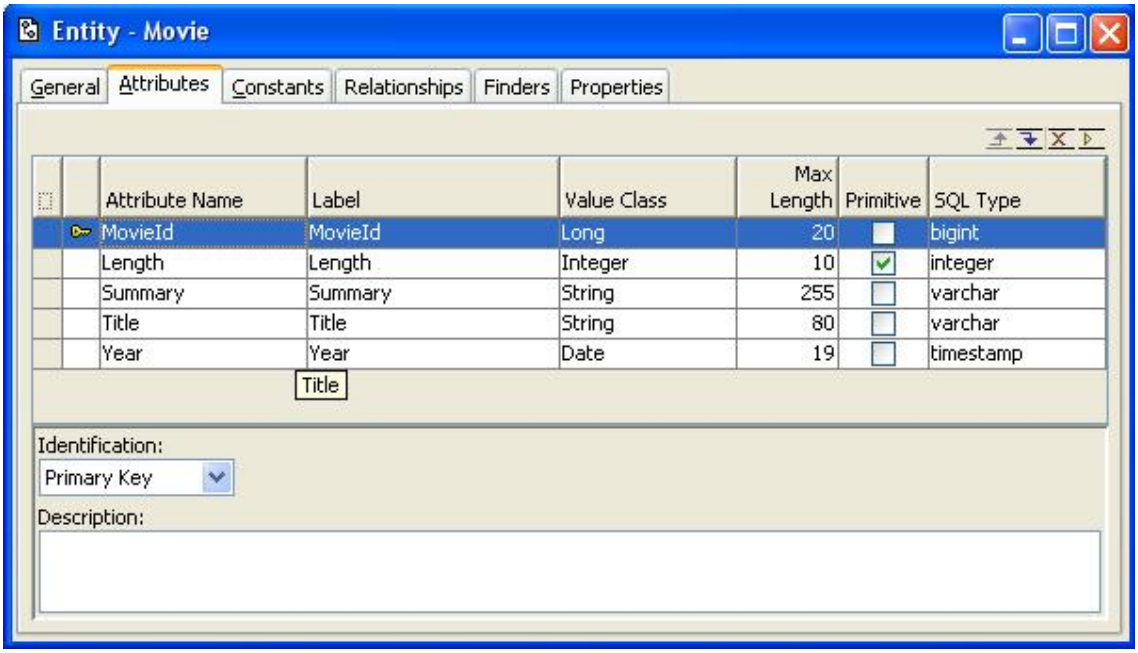

Figure A.6: Importing attributes to an entity

A.3.5 Importing relationships

We have now modeled the two entities and wish to model the relationship between them. To do this we open the object view for either one and choose the tab “Relationships”. In this case we chose the Movie entity. We then proceed by choosing “Tools/Import Table Relationships”. With the guidance of the foreign keys defined in the database DOI Studio will now automatically create and name this relationship. The results of this can be seen in Figure A.7.

As seen in Figure A.7 there are two sides showing similar information. The fields on the left and right sides represent the relationships from the viewpoint of the current entity and the target entity, respectively. This means that we do not need to repeat the process of importing the relationship for the Actor entity. We have also checked the “navigable” checkboxes on both sides of the relation to ensure that methods will be generated in the brokers for accessing related entities.

When we save this view, DOI Studio will have persisted all information about the relationship to underlying storage. The relationship will be represented with methods in the entity beans in the generated code.

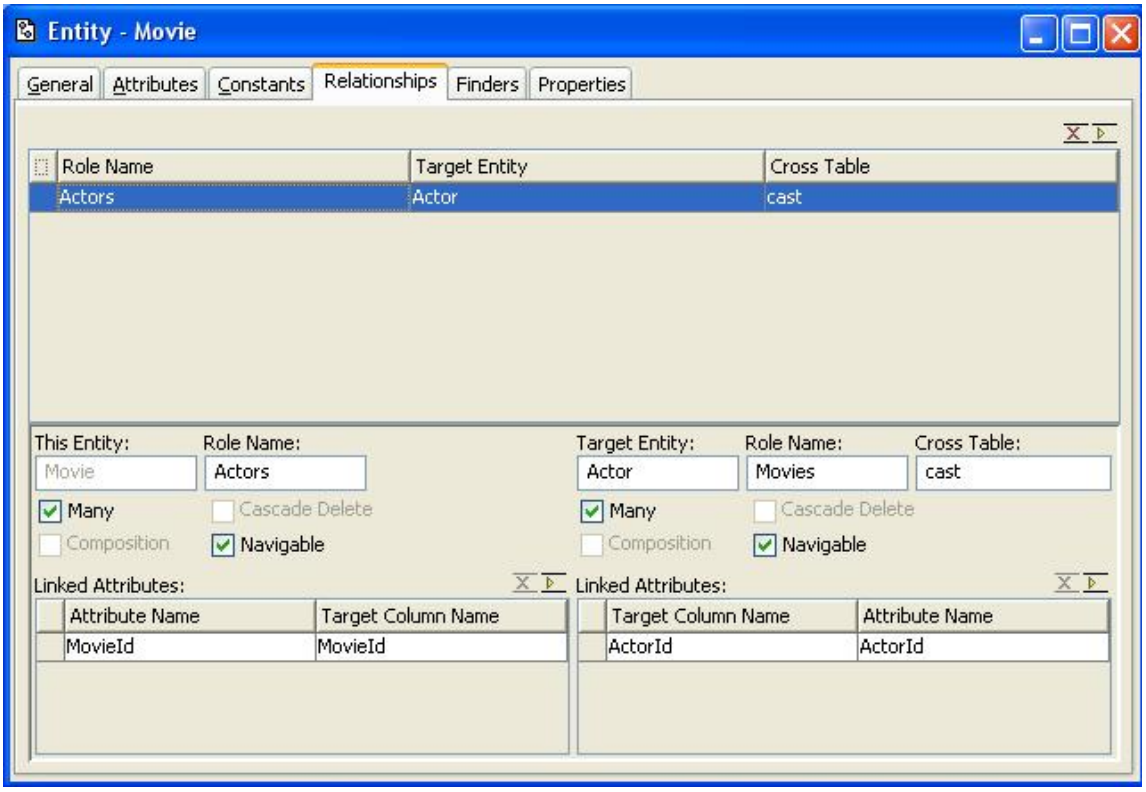

Figure A.7: Relationships between entities in DOI Studio

A.3.6 Generating the project

We are now ready to generate the project. To do this we open the object view representing our project and then choose “Tools/Generation of Project Files”. In the view that appears the correct project code will have already been entered. We also have the option to only generate code belonging to a specific entity. Since in this case we want to generate the entire project this field is left empty.

When we press the “Generate” button we will be presented with a view showing all generated files, as seen in Figure A.8. The status field gives us information about whether a file is new, updated or manually edited outside of DOI Studio. We also have an option of whether to write each individual file or not. In this case all files are new so we will proceed with writing them all by pressing the “Write” button. When this is done the status of all files will change to “Not Changed”.

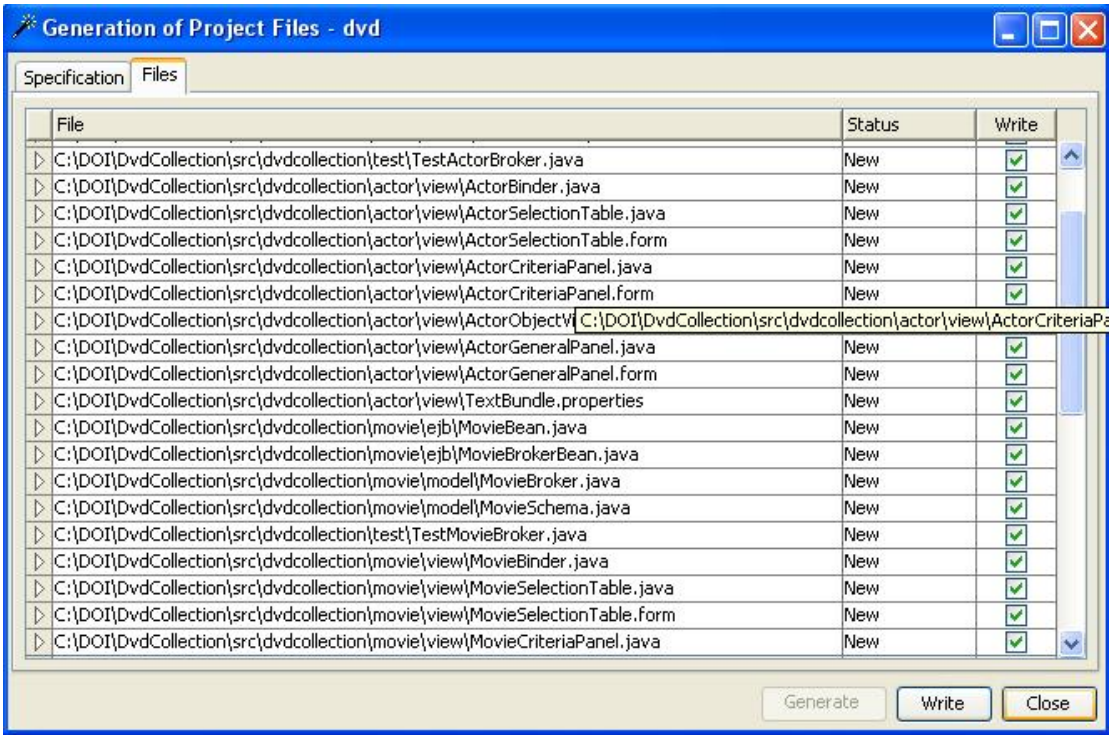

Figure A.8: Generating files in DOI Studio

We now have a complete NetBeans project in the directory provided when creating the project in DOI Studio. At this stage development continues by writing business logic and creating user interfaces in NetBeans.

A.4 Developing in NetBeans

When the project is opened in NetBeans we see that DOI Studio has generated object views for both the Movie entity and the Actor entity, together with two empty panels called MovieGeneralPanel and ActorGeneralPanel, respectively. The next step in our example application is to create the Movie user interface. The procedure for creating the Actor user interface is similar and will not be covered.

In this case as in many business applications the user interfaces will consist of components such as combo boxes and text fields, where values can be entered for a specific entity. Besides this basic representation of a movie object we also wish to show the relationship that this movie object has to certain actors. We will do this by providing in the object view two panels: one for general information about a movie object and one for displaying actors related to the specific movie object.

A.4.1 Creating the movie “General” panel

To implement the basic functionality for the Movie entity we open the file MovieGeneralPanel.java in NetBeans. This will show the panel in design mode and allow us to add graphical components from the toolbar. In this example we will only use the component DoiTextField to allow for input of the values. DoiTextField is a flexible component that can represent an ordinary text field but also multi-line fields and combo-boxes. In this example we add four text fields to the panel by dragging and dropping them from the toolbar. When this is done all we have to do is to set a few properties of these text fields. The first one is the property “Name”, which should be set to the string <entity name>.<field name> (e.g. Movie.Title). We also want to set the label text and the value class of the fields. The value class of a field determines which type it is meant to represent (e.g. String, Integer, Date) and needs to be set to correspond with the type of the field in the entity bean.

These simple steps are all that is needed to create a complete (although simple) user interface for the movie object. If we compile and run the application at this stage we will be able to create new Movie entities which will be persisted to the database. Figure A.9 shows the movie interface within the running application.

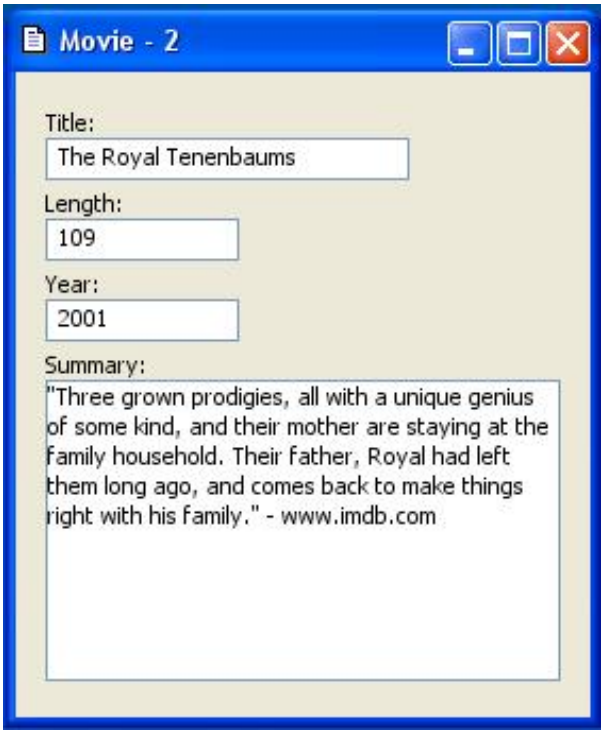

Figure A.9: The movie "general" panel

A.4.2 Creating the movie "Actors" panel

In the final step of this guide we will extend the user interface to also represent the relationship between the Movie and Actor entities. To do this we create a new class that inherits from DoiObjectPanel and call it MovieActorsPanel. Then, in the graphical design view, we drop a table onto this panel from the DOI components available in NetBeans. This table will have a single column that displays to the user the names of all actors that are related to the current movie. The "name" property of this table is set to "Movie.Actors", which is the name of the relation. The next step is to add a table column to the table by dragging it into the table in the graphical designer. The "name" property is set to "Actor.Name", which is the field in the Actor entity that we wish to show for all actors related to a movie. This concludes the creation of the "actors" panel. The final step is to add the panel to the user interface for the movie entity. This is done by adding a line of code in MovieObjectView.java, as shown in Figure A.10. Figure A.11 shows the panel in the running application.

```
/**
 * Construct a new object view for the Movie binder.
 * @param pBinder The binder.
 */
public MovieObjectView(MovieBinder pBinder)
        throws IOException
{
        super(pBinder);

        addObjectPanel(new MovieGeneralPanel(this));
        addObjectPanel(new MovieActorsPanel(this));  // added line
}
```

Figure A.10: Adding the new panel to the object view

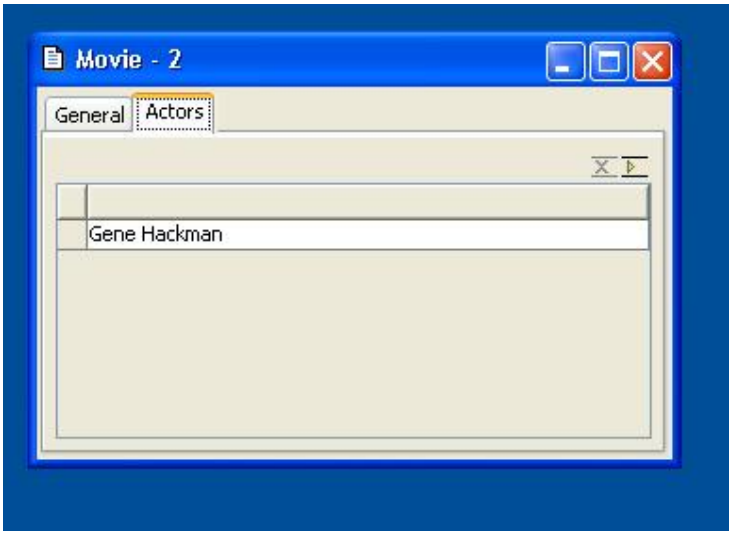

Figure A.11: The movie "actors" panel

A.5 The finished application

Figure A.12 shows the finished application running. New movies can be added by choosing the movie node in the tree and pressing the “New” button in the menu. To display all movies in a selection table double-click the “Movies” node in the tree. It is also possible to navigate from a movie object view to an actor object view, through the “Actors” panel. This is a few examples of functionality provided by default in DOI applications.

The purpose of this appendix was to explain how the different parts of the DOI framework contribute to the development. For this reason we chose not to describe in detail the creation of further user interfaces such as search criteria, since the process of creating these interfaces differs little from the process discussed for creating the object views.

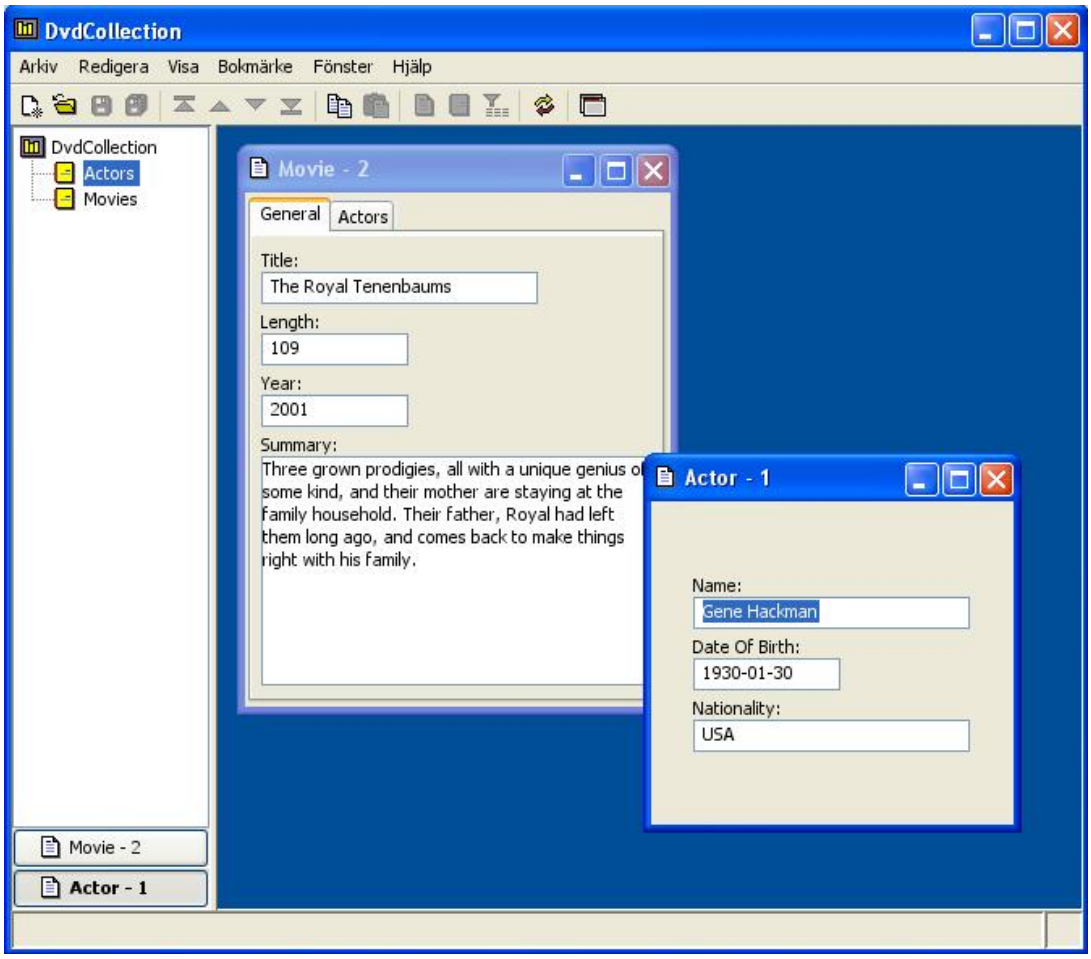

Figure A.12: The finished application