A Reusable Persistence Framework for Replicating Empirical Studies on Data from Open Source Repositories

Scott Bong-Soo Chun

Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

https://scholarsarchive.byu.edu/etd/2371

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen amatangelo@byu.edu.
A Reusable Persistence Framework for Replicating Empirical Studies on Data From Open Source Repositories

Scott B. Chun

A thesis submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of Master of Science

Charles D. Knutson, Chair
Christophe G. Giraud-Carrier
William A. Barrett

Department of Computer Science
Brigham Young University
April 2011

Copyright © 2011 Scott B. Chun
All Rights Reserved
ABSTRACT

A Reusable Persistence Framework for Replicating Empirical Studies
on Data From Open Source Repositories

Scott B. Chun
Department of Computer Science
Master of Science

Empirical research is inexact and error-prone leading researchers to agree that replication of experiments is a necessary step to validating empirical results. Unfortunately, replicating experiments requires substantial investments in manpower and time. These resource requirements can be reduced by incorporating component reuse when building tools for empirical experimentation. Bokeo is an initiative within the Sequoia Lab of the BYU Computer Science Department to develop a platform to assist in the empirical study of software engineering. The i3Persistence Framework is a component of Bokeo which enables researchers to easily build and rapidly deploy tools for empirical experiments by providing an easy-to-use database management service. We introduce the i3Persistence Framework of Bokeo to assist in the development of software to replicate experiments and conduct studies on data from open-source repositories.
ACKNOWLEDGMENTS

First and foremost, to my wife Andrea, who never gave up even when I did.

To my children, Christopher, Samuel, Alexander, Daniela, and Dominik, for their patience when Papa was "too busy."

To my advisor, Dr. Charles Knutson, for lighting a fire under me and making sure I finished.

To my colleague Alex MacLean for his counsel and critique which have improved this thesis.

To Dr. Benjamin Gibbs whose orthogonal view added a new dimension to this thesis.

To my sister-in-law Beth for the time she spent to edit this thesis.

To my co-workers, Ben Wood and Jason Hill, without whose collaboration this project would not have been possible.

To my father who in his dying breath instructed me to pursue higher education.

Thank you.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Role of Replication in Empirical Research</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Bokeo: A Framework for Conducting Empirical Experiments</td>
<td>2</td>
</tr>
<tr>
<td>1.3 The Database Management System</td>
<td>4</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>6</td>
</tr>
<tr>
<td>3 Thesis Statement</td>
<td>8</td>
</tr>
<tr>
<td>4 Project Description</td>
<td>9</td>
</tr>
<tr>
<td>4.1 Why Another Persistence Framework?</td>
<td>9</td>
</tr>
<tr>
<td>4.1.1 Limitations of Existing Database Management Systems (DBMSs)</td>
<td>9</td>
</tr>
<tr>
<td>4.1.1.1 The Relational vs. Object-Oriented Impedance Mismatch</td>
<td>9</td>
</tr>
<tr>
<td>4.1.1.2 The Challenges of SQL</td>
<td>10</td>
</tr>
<tr>
<td>4.1.1.3 The Challenges of Modern Object-Oriented DBMSs</td>
<td>12</td>
</tr>
<tr>
<td>4.1.2 The Traditional vs. i3Persistence Development Process Model</td>
<td>13</td>
</tr>
<tr>
<td>4.1.3 Static vs. Dynamic Database Schema</td>
<td>14</td>
</tr>
<tr>
<td>4.2 i3Persistence Framework User Specifications</td>
<td>15</td>
</tr>
</tbody>
</table>
4.2.1 Decoupling via the *Dependency Inversion Principle* ............... 16
4.2.2 Further Decoupling via the *Dependency Injection Pattern* .......... 16
4.2.3 Delegation via The *Factory Method Pattern* .......................... 17
4.2.4 Reduce Complexity via Query By Filter vs. SQL ...................... 18
4.2.5 Reduce Code via Automatic Generation of *interface* Implementations 18
4.2.6 Ease of Use via a Simple Persistence API .............................. 19
4.2.7 Support for MySQL ............................................... 20

5 Implementation ......................................................... 21
5.1 Architecture Overview ............................................... 21
5.1.1 Project Organization ............................................... 21
5.2 Java Object Persistence Scheme ....................................... 23
5.2.1 Java Instrumentation and *i3Agent* ................................ 25
5.2.2 *Javassist* and Byte Code Manipulation .............................. 26
5.3 *i3Injection*: Java Annotation ....................................... 26
5.3.1 *@ServiceContext* .................................................. 27
      5.3.1.1 Registering Services with *initializationClasses* .............. 27
      5.3.1.2 Controlling the Scope of a *ServiceContext* with *contextType* 28
5.3.2 *@RegisterService* ............................................... 30
5.3.3 *@RegisteredClass* ................................................ 30
5.3.4 *@Service* .......................................................... 32
5.3.5 *@New* .............................................................. 33
5.3.6 *@Transaction* .................................................... 34
5.4 Optimizing Object Injection ........................................... 34
5.4.1 Fine Tuning The Injection with the *injection-conf.xml* File ....... 35
5.4.2 *@DoNotTransform* ................................................ 36
5.5 Framework Services .................................................. 37
5.5.1 The *Factory Service Interface* .................................... 37
9 Future Work

9.1 Point-in-Time Architecture ............................... 68
9.2 Optimize Speed ........................................... 68
9.3 Support Large Datasets ................................... 69
9.4 Object Version Migration ................................. 69

Bibliography

A Separating Responsibilities with a Component-Layered Architecture 75
B Decoupling Modules with the Dependency Inversion Principle 77
C Connecting Components and Layers with the Dependency Injection Pattern 79
D ChangelogExtractor Module injection-config.xml 82
E ChangelogAnalyzer Module injection-config.xml 83
F ChangelogDAOTest.java 84
G ModificationRequestBuilder.java 94
H i3Persistence Framework, JDBC/SQL and Berkeley DB Test Results 97
I Gnu gcc: Distribution of the Number of Files in an MR 98
J Gnu gcc: Modification Requests per Month, 2003 99
K UML Diagrams
   K.1 i3Datasource: Module Statistics and Package Diagram .......................... 100
   K.2 i3InjectionApi: Module Statistics and Class Diagram .......................... 101
      K.2.1 i3InjectionApi: Service Module Class Diagram ............................. 102
K.2.2 i3InjectionApi: Modules Class Diagram ................. 103
K.3 i3PersistenceApi: Module Metrics and Package Diagram .......... 104
  K.3.1 i3PersistenceApi: Query Module Class Diagram .......... 105
  K.3.2 i3PersistenceApi: Modules Class Diagram ................. 106
K.4 i3InjectionImpl: Module Statistics and Package Diagram .......... 107
K.5 i3PersistenceImpl: Module Statistics and Package Diagram .......... 108
List of Figures

1.1 High-level Workflow Phases of Bokeo. ............................ 3

4.1 SQL-based Database Development Process .......................... 14
4.2 i3Persistence-based Database Development Process ............... 15
4.3 i3Persistence-based Plug-in Development Process ................ 16

5.1 i3Persistence Framework Component-Layered Architecture .......... 22
5.2 Java Object Serialization ......................................... 23
5.3 i3Persistence Framework Database ER Diagram ................ 24

6.1 SerialSchema.sql file location .................................... 46
6.2 connection.properties file location ................................ 47
6.3 connection.properties file Settings ................................ 47
6.4 Eclipse Build Path ................................................ 48
6.5 Eclipse Runtime Configuration Classpath ......................... 48
6.6 Eclipse Runtime Agent Configuration Classpath ................ 49

7.1 Software Trails .................................................... 50
7.2 The sliding-time-window algorithm ................................ 51
7.3 ChangelogExtractor module class diagram ......................... 55
7.4 ChangelogAnalyzer module class diagram ......................... 58
7.5 Number of files in MRs ............................................. 61
7.6 MRs per month, 2003 ............................................... 61
List of Tables

4.1 JavaBeans Properties Naming Convention ........................................ 19

5.1 i3Persistence Framework Database Table Definitions .......................... 25
5.2 PersistenceStatus Values ................................................................. 25
5.3 ServiceContext Annotation Optional Element Summary ........................ 27
5.4 ServiceContextType Values ............................................................... 29
5.5 PersistenceManager Method Summary ................................................. 40
5.6 Query Class Method Summary ........................................................... 40
5.7 FilterAdapter Class Method Summary ............................................... 41
5.8 FilterGroupAdapter Class Method Summary ....................................... 41

7.1 ChangelogExtractor module class descriptions .................................... 56
7.2 ChangelogAnalyzer module class descriptions ..................................... 59
7.3 CVS statistics for the GNU gcc project ............................................... 60
7.4 JDBC vs. i3Persistence: Database class .............................................. 63
7.5 JDBC vs. i3Persistence: DAO class ................................................... 64

H.1 Comparison of GNU gcc cvs log Analysis ...................................... 97
I.1 Distribution of the Number of Files in an MR .................................... 98
J.1 Modification Requests Per Month, 2003 ............................................ 99
K.1 i3Datasource Module Metrics ......................................................... 100
K.2 i3InjectionApi Module Metrics ....................................................... 101
List of Listings

4.1 Oracle Debugging Setup ........................................... 11
4.2 Oracle SQL Debugging Code ........................................ 11
4.3 MySQL SQL Debugging Code ........................................ 12
5.1 ServiceContext **Initializer** Class .............................. 27
5.2 JUnit Test for Manual Control of Service Context .............. 29
5.3 LazyInterface example ............................................ 31
5.4 LazyAbstractClass example ....................................... 31
5.5 LazyInterface **Initializer** class ............................... 31
5.6 Instantiating an **abstract class** for Injection ................ 32
5.7 @Service declaration .............................................. 33
5.8 @New declaration .................................................. 33
5.9 @Transaction interface ........................................... 34
5.10 i3InjectionImpl injection-config.xml file ....................... 35
5.11 i3Datasource injection-config.xml file ......................... 36
5.12 @DoNotTransform annotation usage .............................. 36
5.13 Class Instance Factory ........................................... 37
5.14 Custom Factory Implementation .................................. 38
5.15 **Initializer** Class for Custom Factory Implementation .... 38
5.16 Custom Factory Implementation Injection ........................ 38
5.17 interface to Persist ............................................... 42
5.18 **Data Access Object** Class ..................................... 42
5.19 DAOClass **accessObjects**() Method ......................... 43
Chapter 1

Introduction

1.1 The Role of Replication in Empirical Research

Basili [4], Brooks [8], MacDonnell [32] and other researchers [23, 24, 40] state that empirical studies tend to be inexact and error-prone; thus, results need to be validated. Empirical experiments are fraught with potential errors: statistical procedures may be misapplied, the sample size may not be adequate to support the results, and/or the results may not justify the conclusion.

Because of such problems, researchers consider reproducibility of experimental results a prerequisite to establishing validity. Shull et al. state that one of the primary goals of replication is to verify the applicability of the results of the initial study [39].

One method of validating the results of an experiment is for multiple independent teams to conduct the same experiment concurrently [8]. The confirmation of two or more independently conducted experiments adds credibility to the results.

Another way to facilitate replication is to publish detailed reports on how to replicate the empirical experiment. When describing an experiment a researcher provides a recipe for others to verify results by documenting the type of equipment used and the precise procedure followed. Basili et al. suggest a project package structure which outlines the minimal information necessary to facilitate the external replication of an experiment [39].

A third way to facilitate replication is to use experiment databases (ExpDB) to store detailed information on numerous learning experiments [6, 7]. An ExpDB is a special kind of inductive database whose methodology consists of filling and then querying this database
for patterns [37]. This methodology is useful for data mining and machine learning where insights into the behavior of learning algorithms are obtained by implementing the algorithms and studying their behavior on specific datasets. Although ExpDB has been successfully used in empirical experiments in various fields of study (i.e., biology, physics) where algorithms are applied to well-defined datasets, it is not as readily applicable in empirical studies of software evolution because such studies are exploratory in nature and are not as well-defined as learning experiments. Therefore, the ExpDB methodology is mostly applicable to software engineering empirical studies which optimize algorithms that work on well-defined datasets.

Experimental replication may require the researcher to invest substantial manpower and/or time to develop and maintain the software tools necessary to validate empirical experiments.

One technique to reduce the resources required to develop tools and increase experimental accuracy is to incorporate code reuse when implementing the tools. Studies have shown the positive effects of code reuse on reducing the time required to develop and maintain software projects [3, 11, 42]. We intend to apply similar principles to experimental replicability.

We present a reusable persistence framework to help implement and deploy tools for replicating empirical experiments.

1.2 Bokeo: A Framework for Conducting Empirical Experiments

Although tools are available to assist with empirical experiments, we are not aware of any tools with the extensibility and power to conduct a broad scope of empirical software engineering research on a large corpus of data. Most research tools are intended as proof-of-concept and are designed to test the hypothesis of a particular experiment. Such tools usually have limited functionality, applicability, and scalability. Other tools define a static workflow tailored to the specifics of an experiment, making them difficult to extend to new experiments.
Consequently, the Sequoia Lab at BYU is actively developing *Bokeo* to help researchers conduct empirical experiments on large corpora of open source software. *Bokeo* is an extensible framework with which users may extract, analyze and visualize derived metrics and data from software repositories. *Bokeo* follows a conceptual workflow-based model with four distinct areas of abstraction called phases, each comprised of one or more components and workflows.

In this section we present the overall architecture of *Bokeo* and describe the high-level functions of each phase (see Figure 1.1).

![Figure 1.1: High-level Workflow Phases of Bokeo.](image)

The first phase of *Bokeo* is the *Extraction Phase*, which extracts raw data from source code or other artifacts of software and bug repositories. Our goal is to aggregate, into a single repository, evolutionary data from many projects across multiple revision control systems to allow researchers to conduct a broad scope of experiments with varying sample sizes.

The second phase of *Bokeo* is the *Analysis and Synthesis Phase*, which analyzes and synthesizes raw data to generate new, more informative *derived* data. Unlike controlled experiments in which the variation in data is dictated by the design of the experiment, the data contained in software and bug repositories are historical and fixed. The variation in the data is extracted or derived in the *Analysis and Synthesis Phase* for the purpose of testing the hypothesis of the empirical experiment, validating the results of an experiment, or viewing a metric to gain insights into a complex data set. Therefore, the majority of the
work for an empirical experiment will normally be carried out in the *Analysis and Synthesis Phase*.

The third stage of *Bokeo* is the *Aggregation Phase*, which performs the aggregation, correlation, and dissemination of both raw and derived data into forms consumable by the *Presentation Phase*.

The fourth stage of *Bokeo* is the *Presentation Phase*, which allows users to navigate and explore derived data in order to gain insights about large and complex data sets.

### 1.3 The Database Management System

The study of software engineering allows researchers to create better models of software processes, products, and environmental factors and to understand how these models interact. This knowledge, in turn, empowers practitioners to better control the development process by understanding what techniques work and what techniques do not work, and under what conditions.

Most artifact-based empirical studies of software engineering analyze one or more metrics extracted from the software or bug repository. As an example, in the case of software evolution, this metric may be compared at different points in the life of the software project to identify trends or patterns of behavior. For example, Gall et al. used software change version information, fine grained changed entities data and Concurrent Versions System (CVS) release history data to create models of (1) project growth and change behavior, and (2) the dependencies within the evolution of particular entities [15].

A central requirement when conducting artifact-based empirical experiments of open source software development for academic research is to share and store comparable data and analysis of software artifacts [22, 17]. The most common method of storing and organizing data is the use of a database management system (DBMS). *i3Persistence Framework* upon which *Bokeo* is being built, reflects the current state of the art in database ease of use and
setup. This persistence framework will allow researchers to rapidly develop and deploy Bokeo plug-in modules.

The *i3Persistence Framework* was originally designed as a standalone technology to address the general need in the software development community to increase developer productivity and decrease the cost of software maintenance. However, in this industrial thesis, we focus on the applicability of the *i3Persistence Framework* in replicating artifact-based empirical experiments on data from open source repositories.
Chapter 2

Related Work

The analysis and synthesis of data from source code and bug repositories is an increasingly well-researched problem with tools emerging within the open source community. These tools support the extraction of data from repositories and the generation of high-order, derived metrics via analysis and synthesis techniques. In this section we present relevant related work.

Bonsai [21] and LXR [19], two commonly used CVS front-end tools, store the revision information for each file into a relational database. Both Bonsai and LXR consist of several PERL scripts that are run in succession. Although many researchers have managed to generalize Bonsai and LXR, their maintainers recommend using these tools with the repository for which they were originally designed, namely Mozilla and the Linux kernel respectively. Bonsai and LXR are characteristic of tools that are designed to work on a narrow group of repositories but which researchers have adapted for their particular empirical needs.

SoftChange tracks changed files in C/C++ or Java software projects by aggregating data from the CVS source repository, Bugzilla bug repository and mail archives [18]. This tool consists of several PERL scripts designed to run in sequence on a Linux platform. SoftChange requires the user to install an instance of the PostgreSQL database into which intermediary information is placed. SoftChange is not designed for use in an automated workflow scenario due to technology dependencies, as well as required human interaction.

Hipikat identifies relationships between documents by aggregating information from CVS source and Bugzilla repositories, newsgroups and archives of mailing lists [10]. The
goal of Hipikat is to recommend to the user existing artifacts from a software project that are relevant to a task of interest. This tool is implemented as a plug-in to Eclipse, and hence cannot be used as a standalone contributor in an automated workflow.

Xia/Creole is an Eclipse plug-in which leverages the Eclipse CVS API and the Shrimp Visualization Tool [41] to create a visual architectural representation of CVS repositories [44, 30]. Since Xia/Creole is implemented as a plug-in, it cannot be used as a standalone component in an automated workflow.

The Google App Engine is an end-to-end Web application development and runtime environment. Developers build App Engine applications using the provided SDK and deploy them onto Google’s infrastructure. For the App Engine datastore, the Java SDK includes implementations of the Java Data Objects (JDO) and Java Persistence API (JPA) interfaces. The current App Engine JDO implementation does not support inheritance, and only objects in the same group can be saved in one transaction. Additionally, the App Engine incorporates a pay-as-you-go fee model: the application resources used, such as storage and bandwidth, are measured and billed by the gigabyte.

Although analysis and data transformation tools have been developed, much work remains in improving the extensibility, reusability, and scalability of such tools. The most significant limitations of existing tools are that (1) they require complex setup rendering them difficult to utilize, (2) they enforce a static workflow which is not adequate for general use, (3) they are designed with non-standard interfaces, making them difficult to reuse in a larger tool context, and (4) they are generally not scalable under increased load.
Chapter 3

Thesis Statement

In this industrial thesis, we present a reusable persistence framework, called i3Persistence, to increase developer productivity in replicating data-centric empirical experiments by abstracting away from the user the complexity of data management, and facilitating ease of use and setup. We validate this framework by replicating the GNU gcc portion of German’s study, Mining CVS Repositories, the softChange Experience, using the Sliding-Time-Window algorithm to rebuild Modification Requests from CVS revision logs.
Chapter 4

Project Description

4.1 Why Another Persistence Framework?

Given the plethora of relational database management systems (RDBMS) currently available, one may argue against the need for yet another persistence framework. We present our justification below in three parts:

1. The limitations of existing database management systems.
2. The deficiencies of data management in the current development model for empirical experiments of open source software.
3. The lack of support for Bokeo’s plug-in architecture requirements by the design-time static schema requirement of current RDBMSs.

4.1.1 Limitations of Existing Database Management Systems (DBMSs)

Although Database Management System (DBMS) technology has evolved from the tightly integrated systems of the mid 1960s to the cloud-based systems of today, the basic building blocks of general-purpose DBMSs continue to pose obstacles to the application developer. We outline these obstacles in the following subsections.

4.1.1.1 The Relational vs. Object-Oriented Impedance Mismatch

Most applications that rely on a database as a datastore use a RDBMS while employing an object-oriented programming language to implement business logic. Since RDBMSs store
data in tables which is inconsistent with this programming model, business objects must be mapped to and from predefined database tables. This inefficiency, or “impedance mismatch,” resulting from the need to map objects to tables and vice versa, has been a necessary complexity and performance penalty when using RDBMSs.

4.1.1.2 The Challenges of SQL

The use of RDBMSs requires proficiency in SQL (Structured Query Language), the most widely used computer language designed for managing data in RDBMSs. Unfortunately, the use of SQL poses several challenges to the developer.

First, although there is an accepted SQL standard, there are significant differences in DBMS implementations [2]. These differences are significant to the degree that SQL written for one DBMS will often not work on another DBMS. The reason for this can be traced to the fact that most modern database systems existed long before SQL was standardized and that there is still a business need to continue to support the legacy SQL written for these distinct environments. This uneven implementation of SQL and proprietary SQL extensions among RDBMS products makes migrating a sizable application from one RDBMS to another a significant effort.

Second, the grammar of SQL is based on declarative programming as opposed to imperative programming paradigm of traditional computer programming languages. In declarative programming, computation is described in terms of what the program should accomplish without prescribing the details of the actions to be taken. In contrast, imperative programming describes how to accomplish a task in terms of specific directions for the computer to perform. This difference in programming paradigms imposes another “impedance mismatch” for developers.

Third, although there have been great strides made in the development of database debuggers, these tools usually support a limited set of databases and require intimate knowledge of the particular flavor of SQL supported by each database. This results in an obtrusive
“trial-and-error” development process which usually requires substantial time and resources (see Section 4.1.2). Additionally, a debugging solution for one RDBMS may not work for other vendors’ products. Such is the case for Oracle and MySQL.

For example, the accepted practice when debugging Oracle queries is to use the 
\texttt{DBMS\_OUTPUT} package. This package is a corollary to \texttt{System.out.println()} in Java. The developer inserts output statements at strategic points in the code during design to evaluate the state of the executing query.

Additionally, in order to view the output at the command-line in Oracle without using a database-aware debugger such as SQL*Developer or Toad, the developer must first set a SQL*Plus environment variable from inside the SQL*Plus command-line environment (Listing 4.1). The developer can then test anonymous or named blocks (which is not supported in MySQL) with SQL statements such as in Listing 4.2.

\begin{verbatim}
Listing 4.1: Oracle Debugging Setup
SQL> SET SERVEROUTPUT ON SIZE 1000000
\end{verbatim}

\begin{verbatim}
Listing 4.2: Oracle SQL Debugging Code
-- Create a procedure in Oracle.
CREATE OR REPLACE PROCEDURE hello_world IS
BEGIN
  -- Print a word without a line return.
dbms_output.put('Hello ');
  -- Print the rest of the phrase and a line return.
dbms_output.put_line('World!');
END;
/
EXECUTE hello_world;
\end{verbatim}

The MySQL equivalent to calling the \texttt{DBMS\_OUTPUT} package procedures is to simply \texttt{SELECT} a string as in Listing 4.3.

\begin{verbatim}
-- Call the procedure.
EXECUTE hello_world;
\end{verbatim}
As can be seen in Listing 4.2 and Listing 4.3, the syntax of the SQL statements (as well as the APIs used for debugging) substantially differ between Oracle and MySQL. This example illustrates that debugging knowledge and efforts for Oracle are not transferable to MySQL and vice-versa.

4.1.1.3 The Challenges of Modern Object-Oriented DBMSs

An alternative to RDBMS is to manage data through object-oriented database management systems (OODBMS). There are many products currently available and most have semantics that are close to those of object-oriented programming languages, making it relatively easy to store, retrieve, and manage objects rather than rows of data. However, although the acceptance of OODBMS is growing, this solution also has its drawbacks.

While a standard exists for OODBMSs [5], in practice most products implement very little of that standard and vendors differ broadly on which portions of the standard they do implement. Additionally, most OODBMSs require knowledge of OQL, in addition to SQL, to manipulate the data. Due to the complexity of OQL, many OODBMS vendors implement
only subsets of the standard, leading to the same diversity problem of SQL described in Section 4.1.1.2.

4.1.2 The Traditional vs. i3Persistence Development Process Model

Most existing DBMSs are designed for general use and are not optimized for computation-centric research. One of the building blocks of a general application DBMS is the Data Manipulation Subsystem, which helps the user to add, change, and delete information in a database and query it for valuable information. To maximize flexibility, most RDBMS Data Manipulation Subsystems support the use of a Data Query Language called SQL.

Complex computations are written in SQL and executed on the database server. SQL is complex to write and difficult to debug, thus increasing the likelihood of defects and prolonged development time.

The standard process for developing complex SQL logic is to implement the code in steps where the outputs of each coding step are saved to be analyzed at a later time for correctness. This process is repeated until the desired results are generated. This “non-runtime” debugging process, as depicted in Figure 4.1, usually involves the use of multiple tools and significantly more iterations than “runtime” debugging processes of modern programming languages.

The i3Persistence Framework alternative is to write the computations and business logic in Java and use a persistence-only optimized system to handle the storing and retrieving of data. This process is shown in Figure 4.2.

In this development model, computations are written in Java using tools with which the developer is familiar and which support a robust write-debug-modify cycle. The complexity of storing and retrieving data is abstracted behind an easy-to-use interface, freeing developers to focus on the solution to the research question instead of on the details of persistence implementation. Advanced Integrated Development Environments (IDE), such as
Eclipse, allow rapid implementation cycles by supporting code editing and debugging as a contiguous process within a single tool.

4.1.3 Static vs. Dynamic Database Schema

RDBMSs enforce the use of a design-time static schema, which is contrary to Bokeo’s plug-in architecture, in which each plug-in designer can specify persistence requirements. The RDBMS-persistence-based prototype-to-production development model usually requires database changes to be migrated from the prototype to production runtime environment at design time. This implies that the framework must be aware of each plug-in’s persistence requirements.

The goal of Bokeo is to allow researchers to prototype their Analysis and Synthesis Logic outside Bokeo then migrate the modules into Bokeo by wrapping them in the plugin framework as shown in Figure 4.3.

Bokeo provides the same i3Persistence services used during the prototyping phase of development where the Analysis and Synthesis Logic is developed as a standalone application. By using Dependency Inversion (Appendix B), Dependency Injection (Appendix
Figure 4.2: *i3Persistence*-based Database Development Process

C), and *Byte Code Manipulation* (Section 5.2.2), *Bokeo* seamlessly supports the persistence requirements of the plug-in. Therefore, an *i3Persistence Framework*-based standalone application can be converted into a *Bokeo* plug-in without any changes to the persistence logic.

### 4.2 *i3Persistence Framework* User Specifications

The *i3Persistence Framework* was developed over a period of 1.5 years. The primary product goals of the initial release of the *i3Persistence Framework* are three-fold:

1. Allow development teams of various skill levels to work concurrently on the project, but independent of each other.
2. Increase developer productivity by abstracting the complexity of data management away from the user.
3. Promote ease of use and setup.

The following sections describe features that were designed to address the product goals listed above.
4.2.1 Decoupling via the Dependency Inversion Principle

In 1996, Robert Martin postulated the Dependency Inversion Principle (DIP) articulating the need to reduce the dependency of higher-level, more general layers to lower-level, more concrete layers in software systems [33]. This decoupling is achieved through the use of abstractions or, in the case of Java, interfaces.

The i3Persistence Framework achieves reduction in coupling by using a Component-Layered Architecture (see Appendix A). Each layer exposes a Service Interface which abstracts the layer’s implementation details. This decoupling facilitates the independent, concurrent development of the i3Persistence Framework components and layers.

4.2.2 Further Decoupling via the Dependency Injection Pattern

Dependency Injection (DI) is a programming pattern that separates application code from the concrete implementation of the Service Interface. The main idea behind DI is that if an object relies on other "components" to help it accomplish its work, it should not be responsible for creating those components; rather, the components it depends on should be injected into it in the form of abstractions. Therefore, DI is based on the Dependency Inversion concept of programming to abstractions.
One benefit of using DI is the reduction of boilerplate code in the application objects since all work to initialize or set up dependencies is handled by a provider component [27].

Another benefit of DI is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This means that service implementations can be easily changed by simply changing a configuration file.

The *i3Persistence Framework* uses DI to “glue” its services to application logic (see Appendix C).

### 4.2.3 Delegation via The Factory Method Pattern

One of the fundamental goals of software development, as stated by the open/close principle, is to design systems in which software entities (classes, modules, functions, etc.) are “open for extension, but closed for modification” [34]. Simply stated, software systems should be designed to change rapidly while reducing the negative impact these alterations can bring.

The *Dependency Inversion Principle* and *Dependency Injection Pattern* presented previously reduce coupling which in turn reduces the impact of changes to the system, but the use of abstractions introduces another problem. A module at runtime may only know when to instantiate a new object, but not the kind of subclass to create.

A solution to this problem is a creational design pattern called the *Factory Method Pattern* [16]. This pattern addresses the problem of creating objects without specifying the exact class of object that will be created. The classic definition of the *factory method* handles the problem of anonymous object creation by defining a separate method for creating the objects, of which subclasses can then override to specify the derived type of product that will be created.

Another definition of the term *factory method* refers to a method of a *factory* whose main purpose is the creation of objects. The *i3Persistence Framework* implements this definition by providing a mechanism to define an *object factory* with the use of the `@Service`
annotation and by implementing the Factory interface (see Section 5.5). This delegation of responsibility increases application maintainability.

Another benefit of using the Factory Method Pattern is the reduction of boiler-plate code in the application objects since all work to instantiate objects is handled by the i3Persistence Framework.

4.2.4 Reduce Complexity via Query By Filter vs. SQL

The i3Persistence Framework Query object eliminates the need for SQL by supporting query by filter. A Filter Object for a query is used to limit the results returned by a query based upon the return value of a getter method (see Section 5.5.3). For filtering on multiple properties, multiple Filters can be applied to a Query using a FilterGroup.

4.2.5 Reduce Code via Automatic Generation of interface Implementations

To reduce lines of code, the i3Persistence Framework provides a mechanism to automatically generate implementations for interfaces which adhere to the JavaBeans properties naming convention. JavaBeans is a framework for defining reusable modular software components. The class properties must be accessible using get, set, is (not that common) and other methods (so-called getter methods and setter methods), following a standard naming convention [12]. These conventions make it possible to develop tools that can use, reuse, replace, and connect JavaBeans. The JavaBeans properties naming convention is depicted in Table 4.1.

A bean defines a property $p$ of type $T$ if it has getter methods that follow these patterns (if $T$ is boolean, a special form of getter method is allowed):

JavaBeans-compliant classes are used frequently as Data Transfer Objects (DTO). A DTO (formerly known as Value Objects) [14], is a design pattern used to transfer data between software application subsystems. DTOs are often used in conjunction with Data Access Objects (DAO) to retrieve data from a database. The difference between DTOs and
<table>
<thead>
<tr>
<th>Property</th>
<th>Naming Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getter</td>
<td>public T getP()</td>
</tr>
<tr>
<td>Boolean getter</td>
<td>public boolean isP()</td>
</tr>
<tr>
<td>Setter</td>
<td>public void setP(T)</td>
</tr>
<tr>
<td>Array getter</td>
<td>public T[] getP()</td>
</tr>
<tr>
<td>Element getter</td>
<td>public T getP(int)</td>
</tr>
<tr>
<td>Array setter</td>
<td>public void setP(T[])</td>
</tr>
<tr>
<td>Element setter</td>
<td>public void setP(int, T)</td>
</tr>
</tbody>
</table>

Table 4.1: JavaBeans Properties Naming Convention

DAOs is that a DTO does not have any behavior except for storage and retrieval of its own data (getters and setters).

In a traditional three-tiered architecture, DTOs serve dual purposes: first, they provide serializable solutions for objects which are not serializable; second, they implicitly define an assembly phase where all data to be used by the view are fetched and marshaled into the DTOs before returning control to the presentation tier. A benefit of controlling the synthesis of DTOs in the framework is that the object assembly phase can be optimized for a particular platform or database without affecting the client application.

The i3Persistence Framework implementation auto-generation mechanism creates a DTO class definition for an interface at runtime using the Javassist APIs. The fields of the generated DTO are based on the property name p and type T required by the JavaBeans naming convention for getter/setter methods.

This auto-generation mechanism reduces boilerplate code in the application since the framework generates the implementation for an interface. The usage details of this feature are described in Section 5.3.5.

4.2.6 Ease of Use via a Simple Persistence API

The Persistence Layer of the i3Persistence Framework is optimized for data storage and retrieval, leaving computations to be written at the Application Layer in Java. This optimization is manifested in the simplicity of the data management API, which consists of a
PersistenceManager to store data and a Query manager to retrieve data. These two API classes hide the complexity of data management and integrate seamlessly into native Java along with the object-oriented programming paradigm.

Because the i3Persistence Framework is written completely in Java, the framework functions as an extension to the Java language and is supported by the same IDE as is used to develop the Analysis and Synthesis Logic. This integral development allows for rapid write-debug-modification cycles.

4.2.7 Support for MySQL

MySQL is an RDBMS released as open source by MySQL AB in 2000. Since then, MySQL has emerged as the preferred open-source database choice for developers [1]. MySQL runs on more than 20 platforms including Linux, Windows, Mac OS, Solaris, HP-UX, IBM AIX, and is currently being used on every continent. The i3Persistence Framework supports MySQL.
Chapter 5

Implementation

5.1 Architecture Overview

The architecture of a software system is the highest-level breakdown of the system into its parts [14]. It embodies the design decisions that are difficult to change over the course of the software development cycle.

In this chapter, we present what we consider to be the important aspects of the $i3Persistence$ Framework architecture.

5.1.1 Project Organization

The $i3Persistence$ Framework is composed of two layers: the Injection Layer and the Persistence Layer (see Figure 5.1). As the names suggest, the Injection Layer encapsulates the DI functionality and the Persistence Layer encapsulates the Data Persistence functionality.

The Injection Layer is comprised of the following modules:

- $i3InjectionApi$ – The service abstraction for the Injection Layer (see Section 4.2.1)
- $i3InjectionImpl$ – The implementation of the $i3InjectionApi$ services (see Section 4.2.1)
- $i3Agent$ – An implementation of the Java Agent module required by the Java Instrumentation mechanism (see Section 5.2.1)
- $Javassist$ – A third-party library for byte-code manipulation
The Injection Layer has a hard binding to several J2EE modules to support deployment on Tomcat and JBoss.

The Persistence Layer is comprised of the following modules:

- i3PersistenceApi – The service abstraction for the Persistence Layer (see Section 4.2.1)
- i3PersistenceImpl – The implementation for the i3PersistenceApi services (see Section 4.2.1)
- i3Datastore – The database-specific modules
5.2 Java Object Persistence Scheme

The i3Persistence Framework persists Java objects in a Relational Database by utilizing Java Serialization. Serialization is the process of converting a set of object instances that contain references to each other into a linear stream of bytes as outlined in Figure 5.2.

The stream functions as a container for the object. Its contents include a partial representation of the object’s internal structure, including variable types, names, and values as outlined in Section 6 of the Sun Microsystems Java Object Serialization Specification rev. 1.4.4 [28].

The Persistence Layer services an object persistence request by generating a transient encoding of the object via the Java Serialization API. The framework then processes the serialized information modifying it for persistence and retrieval, then passes the modified information to the i3Datastore module. The i3Datastore module persists the object information in the schema depicted by Figure 5.3.

Java objects are broken down into their basic members and stored in the database tables listed in Table 5.1.

Each unique class is stored in JAVA_CLASS. All object instances are registered in the JAVA_OBJECT table along with their associated class name. For each field of an object, an entry is created in JAVA_REFERENCE to record the object hierarchy and in JAVA_OBJECT_FIELD
to record the data. The Persistent_Status field values of the JAVA_OBJECT table determine how long an object is retained by the PersistenceManager and determines if the object is visible to queries. Table 5.2 contains a list of the possible PersistentStatus values.

Object data is stored by type in the remaining tables. All String data are stored in the JAVA_STRING table to optimize searches.

The i3Persistence Framework abstracts the physical representation of the data store in the i3Datastore module with the intent to replace the current class getter-based query filtering with a more performant implementation and to eventually support multiple data sources in various formats and technologies.
Table 5.1: i3Persistence Framework Database Table Definitions

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA_CLASS</td>
<td>One entry for each unique class.</td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td>One entry for each persisted object.</td>
</tr>
<tr>
<td>JAVA_OBJECT_FIELD</td>
<td>One entry for each field of the object. If the object is immutable, the</td>
</tr>
<tr>
<td></td>
<td>value is converted to a string and stored here.</td>
</tr>
<tr>
<td>JAVA_REFERENCE</td>
<td>Object references. This table encapsulates the structure of the object.</td>
</tr>
<tr>
<td>JAVA_OBJECT_WRITE_DATA</td>
<td>Manually serialized by data</td>
</tr>
<tr>
<td>JAVA_BINARY</td>
<td>Binary data such as images</td>
</tr>
<tr>
<td>JAVA_PRIMITIVE_ARRAY</td>
<td>Primitive arrays are serialized data</td>
</tr>
<tr>
<td>JAVA_ARRAY</td>
<td>Registry of complex object Arrays</td>
</tr>
<tr>
<td>JAVA_ARRAY_ELEMENT</td>
<td>Elements of Arrays stored in JAVA_ARRAY</td>
</tr>
<tr>
<td>JAVA_PROXY</td>
<td>Registry of the class component of the Java proxy class instance</td>
</tr>
<tr>
<td>JAVA_PROXYINTERFACE</td>
<td>Registry of the interface component of the Java proxy class instance</td>
</tr>
<tr>
<td>JAVA_ENUM</td>
<td>The static portion of a type Enum. The class component is registered in</td>
</tr>
<tr>
<td></td>
<td>JAVA_OBJECT.</td>
</tr>
<tr>
<td>JAVA_STRING</td>
<td>All string data is stored here to support fast searches.</td>
</tr>
</tbody>
</table>

Table 5.2: PersistenceStatus Values

<table>
<thead>
<tr>
<th>Enum Constant Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>hiddenDependent</td>
</tr>
<tr>
<td>An object with a persistence status of hiddenDependent is retained as long as it is referenced by an object with primary status, either directly or through a chain of other persistent objects.</td>
</tr>
<tr>
<td>primary</td>
</tr>
<tr>
<td>An object with a persistence status of primary is retained until it is explicitly removed from the persistence manager.</td>
</tr>
<tr>
<td>visibleDependent</td>
</tr>
<tr>
<td>An object with a persistence status of visibleDependent is retained as long as it is referenced by an object with primary status, either directly or through a chain of other persistent objects.</td>
</tr>
</tbody>
</table>

5.2.1 Java Instrumentation and i3Agent

Java 5 introduced the Java Instrumentation mechanism, which allows developers to provide Java Agents that can inspect and modify the byte-code of the classes as they are loaded. A Java Agent is a pluggable library that runs embedded in a JVM and intercepts the class loading process.

The i3Agent module of the i3Persistence Framework provides a custom Transformer which is utilized by the Java Agent to support byte-code manipulation and object injection.
5.2.2 Javassist and Byte Code Manipulation

Javassist (Java Programming Assistant) is a class library for the runtime editing of byte-codes in Java [9]. It enables Java programs to define a new class at runtime and to modify a class file when the JVM loads it. The byte-code-level API allows users to directly edit a class declaration at runtime to add new methods.

Javassist also supports runtime reflection by enabling Java programs to use a metaobject that controls method calls on base-level objects.

The Injection Layer of the i3Persistence Framework uses Javassist for service injection.

5.3 i3Injection: Java Annotation

An annotation, in the Java programming language, is a special form of syntactic metadata that can be added to Java source code [25]. Classes, methods, variables, parameters and packages may be annotated. Unlike Javadoc tags, Java annotations can be reflective in that they can be embedded in class files generated by the compiler and may be retained by the Java VM to be made retrievable at run-time [20].

Annotations are used by the i3Persistence Framework to achieve the following:

1. Applying behaviors to user-defined classes and methods.
2. Used by the Injection Layer for service injection.
3. Used by the framework Java Agent hook to transform objects at runtime.

The Transformer object inserted into the VM at application startup by the i3Agent module, using the Java Automation API (see Section 5.2.1), intercepts the Classloader at runtime. It then scans the target class to load using Java reflection for supported annotations.

The following sections describe the custom annotations supported by the i3Persistence Framework.
5.3.1 @ServiceContext

@ServiceContext is a method-level annotation used to create a context within which registered objects and services are available for injection into method variables. When the Transformer detects the @ServiceContext annotation at class loading time, it manipulates the byte-code of the containing object and inserts additional code to the annotated method to access the service registry. This byte-code manipulation is achieved using the Javassist API (see Section 5.2.2).

Table 5.3 details the options for the @ServiceContext annotation.

<table>
<thead>
<tr>
<th>Optional Element Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contextType</code></td>
</tr>
<tr>
<td><code>createOnlyIfNotExists</code></td>
</tr>
<tr>
<td><code>initializationClasses</code></td>
</tr>
</tbody>
</table>

Table 5.3: ServiceContext Annotation Optional Element Summary

5.3.1.1 Registering Services with initializationClasses

The initializationClasses option takes a list of comma-delimited runtime class objects to use to register injection implementations. These injection initialization classes (IIC), or Initializer classes, contain implementation details used by the Transformer to instantiate objects for injection. Listing 5.1 is an example of an injection initialization class.

```
Listing 5.1: ServiceContext Initializer Class

1  public class Initializer {
2    3     /**
4         * 1) Register the TestObject interface so that it can be used
5         *     with the @New annotation.
6         */
7     @RegisteredClass
8     private final Class<TestObjectIntf> testObjectClass = TestObjectIntf.class;
9
10    /**
11       * 2) Register the TestService service so that it can be used
12       *     with the @Service annotation.
13    */
14
15}
```

27
The Initializer class of Listing 5.1 registers three injection implementation references annotated with either @RegisteredClass or @RegisteredService. Sections 5.3.3 and 5.3.2 cover these annotations in detail. Each reference maps a left-hand side (LHS) type with a right-hand side (RHS) implementation. The variable name is used only to be syntactically correct. When the Transformer encounters an injection annotation (@Service or @New), it looks for the LHS type in the service registry of the current service context. If the type exists, it retrieves the RHS implementation details.

If the RHS is an interface (line 8), the default factory generates an in-memory implementation based on the interface’s JavaBeans-compliant getter/setter methods.

If the RHS is an object (line 15), the Transformer uses the object as the implementation to inject.

The use of the Factory interface in line 22 is detailed in Section 5.5.1.

A service context may have only one implementation mapped to each Service Interface. Two different injection sites can receive different implementations for the same Service Interface if the injection sites are bound to different service context initializations (see Section 5.3.5 for details).

### 5.3.1.2 Controlling the Scope of a ServiceContext with contextType

The lifespan or scope of a service context is controlled using the contextType option shown in Table 5.4.
The default value, `standard`, limits the scope of the service context to the target method. When the method exits, the service context is destroyed. This setting is useful in Java SE, single-threaded environments where the scope of a context is tied to an execution thread. When the execution terminates, the service context is no longer needed and is destroyed.

The options `containerContextInitializer` and `containerContextFinalizer` allow developers to manually designate the beginning and ending of a service context. These settings are useful in J2EE, multi-threaded and asynchronous environments where program execution can be distributed across multiple independent processes.

Listing 5.2: JUnit Test for Manual Control of Service Context

```java
public class Test {
    /**
     * Create a service context
     */
    @ServiceContext(contextType = ServiceContextType.containerContextInitializer)
    @BeforeClass
    public static void setupBeforeClass() throws Exception {
    }

    /**
     * Delete a service context
     */
    @ServiceContext(contextType = ServiceContextType.containerContextFinalizer)
    @AfterClass
    public static void tearDownAfterClass() {
    }
}
```

Listing 5.2 shows how the `containerContextInitializer` and `containerContextFinalizer` options can be used to extend a service context scope in a JUnit test from the method to the class.
5.3.2 @RegisterService

@RegisterService is a field-level annotation used in Initializer classes to register a service, or a singleton, with the service context (Listing 5.1).

In an Initializer class, when a field marked with the @RegisteredService annotation is written to, the instance (RHS) assigned to the field is registered with the service context. The identifier for the registered service is defined by the LHS declared field type including the type parameters, if any.

When a field that is marked with the @Service annotation is discovered in any class, and the declared type of the field (including the type parameters, if any) exactly matches the declared type of the @RegisteredService field in the Initializer class, then the instance assigned to the @RegisteredService field will be injected into the @Service annotated field.

5.3.3 @RegisteredClass

@RegisteredClass is a field-level annotation used in Initializer classes to register interfaces and class implementations for injection. This annotation is a specialized version of @RegisterService designed to simplify the injection of auto-generated implementations.

In an Initializer class, when a field marked with the @RegisteredClass annotation is written to, a default factory is generated to synthesize the implementation of the RHS runtime class. If the RHS runtime class is an interface, the default factory will synthesize an in-memory implementation of all non-implemented getters/setters which comply with the JavaBeans naming convention.

An added feature of the @RegisteredClass annotation is that the RHS runtime class can be an abstract class with only the getters/setters of interest implemented. The default factory will auto-generate the remaining getters/setters based on their JavaBeans-compliant method names.
LazyAbstractClass in Listing 5.4 is an abstract class which implements two of the four methods required by LazyInterface of Listing 5.3.

LazyAbstractClass in Listing 5.4 is an abstract class which implements two of the four methods required by LazyInterface of Listing 5.3.

In Listing 5.5, the Initializer class registers LazyAbstractClass as the implementation class for the LazyInterface type.
public class LazyClassInjectionTest
{
    /**
     * Main method for the Service Context
     */
    @ServiceContext(initializationClasses={LazyAbstractClassInitializer.class})
    public void lazyAbstractClassNewTest()
    {
        new Object()
        {
            // Inject the abstract class instance
            @New
            private LazyInterface lazy;

            public void run()
            {
                Assert.assertEquals(lazy.getNames().size(), 0);
            }
        }.run();
    }
}

According to the Java Programming Language Specification, abstract classes cannot be instantiated [20], but the i3Persistence Framework injection mechanism bypasses this restriction by automatically completing the implementation of LazyInterface using Javassist and byte-code manipulation. In Listing 5.6, the i3Persistence Framework transformation mechanism inserts into the LazyAbstractClass declaration at runtime a count field of type int and the getCount()/setCount(int count) getter/setter, effectively completing the implementation of LazyInterface.

The @RegisterClass annotation works in conjunction with the @New field-level annotation to inject implementations into fields (see Section 5.3.5). This feature greatly reduces the lines of code necessary when working with abstract classes.

5.3.4 @Service

@Service is a field-level annotation used to indicate that the marked field must be injected at runtime with a service that implements the functions defined by the declared type of the
A service is a singleton implementation that is registered with a service context via the Initializer class using the @RegisterService annotation (see Listing 5.1).

Fields with the @Service annotations must be populated with an instance of a service before any of the non-static methods of the class (including constructors) are invoked. The service to be injected is identified by the declared type of the field, including the type parameters, if any. For example, when the field declaration in Listing 5.7 is discovered by the Transformer, it will search the service registry of the current service context for an implementation matching the LHS type signature, Factory<MyServiceObject>, to inject.

### Listing 5.7: @Service declaration

```java
@Service
private MyService myService;
@Service
private Factory<MyServiceObject> myFactory;
```

If no implementation exists, then a ServiceNotAvailableException is thrown.

### 5.3.5 @New

@New is a field-level annotation used to mark a variable for injection. Its counterpart, @RegisterClass, is used to register the implementation details (see Section 5.3.3).

When the Transformer detects the @New annotation in Listing 5.8, it searches the service registry in the current service context for an implementation matching the LHS type signature of the marked field.

### Listing 5.8: @New declaration

```java
@New
private LazyInterface lazyObject;
```

This implementation may have been previously registered with the service context using the @RegisteredClass annotation in an Initializer class in the @ServiceContext declaration. If no implementation was registered and if the LHS type is an interface, an
in-memory implementation is auto-generated using the Javassist API. The Transformer then injects the generated object into the annotated field.

This feature reduces the boilerplate code necessary when working with DTOs by delegating the implementation of JavaBeans-compliant interfaces to the framework.

### 5.3.6 @Transaction

@Transaction is a method-level annotation which indicates that the work done by a method should be treated as a single unit of work (a single transaction). If an exception is thrown by the annotated method, then all changes done by the method to the database are undone. In other words, if a method tagged with this annotation throws an exception, then the work done by the method will have no persistent effect on the system. The annotation is defined as follows:

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface Transaction {
    public boolean readOnly() default false;
}
```

The readOnly option ensures that no changes will be committed to the database.

### 5.4 Optimizing Object Injection

The Java Instrumentation mechanism does not discriminate which classes are analyzed for injection. This inefficiency can cause substantial degradation in performance for projects which depend heavily on third-party libraries. Consequently, the i3Persistence Framework provides mechanisms to declare which classes are candidates for transformation.
5.4.1 Fine Tuning The Injection with the injection-conf.xml File

Similar to the connections.properties file discussed in Section 6.1, the i3Persistence Framework searches for an injection-config.xml file at the root of the META-INF directory of every JAR loaded. This XML file contains 1) a list of regular expression filters used to validate the canonical class name of classes to transform and 2) a list of defaultInitializer classes to load when creating a service context. This configuration mechanism is used internally by the i3Persistence Framework to qualify framework classes for transformation and to register internal services. Listing 5.10 is the injection-config.xml file for the i3InjectionImpl module.

Listing 5.10: i3InjectionImpl injection-config.xml file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE injection-config PUBLIC "-//Sequoia Lab//DTD Injection Configuration//EN" "/META-INF/dtd/injection-config.dtd">
<injection-config>
  <!-- Classes to transform -->
  <injection-class-list>
    <injection-class regex="edu\.(byu\.cs\.sequoia\.bokeo)\.*++"/>
    <injection-class regex="org\.(apache\.jsp)\.*_jsp"/>
  </injection-class-list>
</injection-config>
```

The two classpaths under the injection-class-list tag declared in Listing 5.10, edu.byu.cs.sequoia.bokeo.* and org.apache.jsp.*, comprise the system default class transformation filters. Each application with package structures outside those qualified by the system default filters, must contain a connections.properties file to extend the default filter list and qualify the application’s classes for transformation.

Also, developers can simplify the declaration of the @ServiceContext annotation (Section 5.3.1) by registering default initialization classes in the injection-config.xml using the initialization-class-list tag. Listing 5.11 shows the i3Datasource module’s injection-config.xml file with the default list of Initializer classes.
The *Initializer* classes declared in the *injection-config.xml* file are evaluated upon service context creation. The *i3Persistence Framework* uses the *i3Datasource injection-config.xml* file to register default services with every service context.

### 5.4.2 @DoNotTransform

@DoNotTransform is a class-level annotation which developers can use to exempt classes that lie within the scope of the *injection-class* filters from transformation. Listing 5.12 shows how this annotation is used.

```
Listing 5.12: @DoNotTransform annotation usage

@DoNotTransform
class MyClass {
    ...
}
```

The combination of the *injection-config.xml* file and the @DoNotTransform annotation enables developers to control the object injection process of the *i3Persistence Framework*. 
5.5 Framework Services

The i3Persistence Framework provides default services to increase productivity, store objects to the database and query objects from the database. This section introduces these services.

5.5.1 The Factory Service Interface

The Factory interface, in conjunction with the @Service annotation, provides an easy-to-use mechanism to create an instance of the Factory Method design pattern. When the Transformer encounters a field declaration of LHS type Factory and marked with the @Service annotation, it searches the service registry of the current service context for an implementation matching the type Factory and its parameter type(s). If no implementation exists, the framework provides a default implementation which returns an instance of the parameter type.

```
Listing 5.13: Class Instance Factory

1 @ServiceContext
2 public void Test() {
3   @ServiceContext
4     public List<Long> getList(int size) {
5       /**
6         * Inject a default factory which returns the auto-generated
7         * implementation of the TestInterface
8         */
9         @Service
10        private Factory<TestInterface> classFactory;
11       
12       /**
13         * Inject a default factory which returns an instance
14         * of the ArrayList<TestInterface> class
15         */
16       @Service
17       Factory<ArrayList<TestInterface>> listFactory;
18      
19      List<TestInterface> list = listFactory.newInstance();
20      for (int i = 0; i < size; i++) {
21          list.add(classFactory.newInstance());
22      }
23      return list;
24   }
25 }
```

In Listing 5.13, the Transformer first injects into the field classFactory a default factory to generate instances of the wrapper class TestInterface. It then injects into the field listFactory a default factory to generate instances of the ArrayList<TestInterface> class. In line 21 of the listing, an instance of ArrayList<TestInterface> is assigned to the variable list. In the for loop starting at line 22, instances of the wrapper class TestInterface are added to the variable list.

The developer can provide a custom RHS implementation which controls the instantiation of the object. This RHS implementation must implement the interfaces Initializable and the LHS Factory type with the corresponding parameters. Listing 5.14 shows an example of a custom implementation of the Factory interface which returns instances of the ArrayList<Object> class.

```
Listing 5.14: Custom Factory Implementation
1 public class ListFactory implements Initializable, Factory<List<?>> {
2     @Override
3     public List<?> newInstance ()
4     {
5         return new ArrayList<Object>();
6     }
7 }
```

Listing 5.15 is the Initializer class which registers the custom factory implementation for the Factory<List<?>> type with service context.

```
Listing 5.15: Initializer Class for Custom Factory Implementation
1 public class Register {
2     @SuppressWarnings ("unused")
3     @RegisteredService
4     private Factory<List<?>> listFactory = new ListFactory();
5 }
```

In Listing 5.16, the @ServiceContext annotation on line 3 creates a new service context within the scope of the customFactoryImplTest() method.
public FactoryTest
{
  @ServiceContext(initializationClasses={Register.class})
  public void customFactoryImplTest()
  {
    new Object()
    {
      /**
       * Inject the implementation registered in the
       * Initializer class.
       */
      @Service
      private Factory<List<?>> listFactory;
      public void run()
      {
        listFactory.newInstance();
      }
    }.run();
  }
}

During the service context creation process, the framework uses the initializationClasses option to register the ListFactory instance for the type Factory<List<?>> interface. When the Transformer encounters the @Service annotation on line 12, it searches the service registry of the current service context for the implementation for the type with the signature Factory<List<?>> and injects it into the listFactory variable.

The i3Persistence Framework support of the Factory Method Pattern eliminates the need for the boilerplate code necessary when manually implementing the Factory Method Pattern.

5.5.2 The PersistenceManager Service

The PersistenceManager service is a framework service that is used to add objects to the database. Table 5.5 lists the methods supported by the PersistenceManager.

A PersistentID is a Universally Unique Identifier (UUID) guaranteed to be unique. This identifier allows database objects to be shared across instances of the i3Persistence
Framework. The PersistenceManager also manages the object’s PersistentStatus. See Section 5.2 for details on PersistentStatus.

The source listing in Section 5.5.3 shows an example of how to use the PersistenceManager, Query, Filters and FilterGroups.

5.5.3 The Query Interface

Query is a framework class (versus a service) that is used to retrieve objects from the database. A Query instance searches the database for instances of a specific class or its subclasses (polymorphic search). Table 5.6 summarizes the methods of the Query class.

```
<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(java.lang.Object o) Add an object to this PersistenceManager.</td>
</tr>
<tr>
<td>void add(java.lang.Object o, PersistentStatus status) Add an object to this PersistenceManager.</td>
</tr>
<tr>
<td>java.lang.String getPersistentID(java.lang.Object o) Get the persistentID for the given object.</td>
</tr>
<tr>
<td>PersistentStatus getPersistentStatus(java.lang.Object o) Get the persistent status for the given object.</td>
</tr>
<tr>
<td>void remove(java.lang.Object o) Remove an object from this PersistenceManager.</td>
</tr>
</tbody>
</table>
```

Table 5.5: PersistenceManager Method Summary
The results of a Query can be restricted by adding a Filter or a FilterGroup that compares an object’s properties to a set of values using a FilterOperator provided by the Filters and FilterGroups. Filters and FilterGroups, if used, must be added before the execute() method is invoked. FilterAdapter is a default implementation of the Filter interface. Table 5.7 summarizes the methods supported by FilterAdapter.

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterAdapter(java.lang.String propertyName, FilterOperator filterOperator, java.util.Collection&lt;? filterValues)</td>
</tr>
<tr>
<td>Create a new FilterAdapter.</td>
</tr>
<tr>
<td>FilterAdapter(java.lang.String propertyName, FilterOperator filterOperator, java.util.Collection&lt;? filterValues, boolean caseSensitive)</td>
</tr>
<tr>
<td>Create a new FilterAdapter.</td>
</tr>
<tr>
<td>FilterAdapter(java.lang.String propertyName, FilterOperator filterOperator, java.lang.Object... filterValue)</td>
</tr>
<tr>
<td>Create a new FilterAdapter.</td>
</tr>
<tr>
<td>FilterAdapter(java.lang.String propertyName, java.lang.Object filterValue)</td>
</tr>
<tr>
<td>Create a new FilterAdapter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterOperator</td>
</tr>
<tr>
<td>setFilterOperator()</td>
</tr>
<tr>
<td>Gets the filter operator for this filter.</td>
</tr>
<tr>
<td>java.util.Collection&lt;? filterValues)</td>
</tr>
<tr>
<td>setFilterValues()</td>
</tr>
<tr>
<td>Gets the filter values that the property value will be compared against.</td>
</tr>
<tr>
<td>java.lang.String</td>
</tr>
<tr>
<td>setPropertyName()</td>
</tr>
<tr>
<td>Gets the name of the property to filter by.</td>
</tr>
<tr>
<td>boolean isCaseSensitive()</td>
</tr>
<tr>
<td>Gets the case sensitivity flag for this filter.</td>
</tr>
</tbody>
</table>

Table 5.7: FilterAdapter Class Method Summary

A FilterGroup allows multiple Filters to be combined into “AND” or “OR” groupings designated by FilterGroupOperator values (see Table 5.8).

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addFilter(Filter filter)</td>
</tr>
<tr>
<td>Add a filter to this group.</td>
</tr>
<tr>
<td>void addFilterGroup(FilterGroup filterGroup)</td>
</tr>
<tr>
<td>Add a filter group to this group.</td>
</tr>
<tr>
<td>void clearFilters()</td>
</tr>
<tr>
<td>Remove all the filters and filter groups from this query.</td>
</tr>
<tr>
<td>FilterGroupOperator getFilterGroupOperator()</td>
</tr>
<tr>
<td>Gets the filter group operator for this FilterGroup.</td>
</tr>
<tr>
<td>java.util.Collection&lt;? extends FilterGroup getFilterGroups()</td>
</tr>
<tr>
<td>Get the FilterGroup instances for this FilterGroup.</td>
</tr>
<tr>
<td>java.util.Collection&lt;? extends Filter getFilters()</td>
</tr>
<tr>
<td>Get the Filter instances for this FilterGroup.</td>
</tr>
<tr>
<td>void setFilterGroupOperator(FilterGroupOperator filterGroupOperator)</td>
</tr>
<tr>
<td>Set the filter group operator for this group.</td>
</tr>
</tbody>
</table>

Table 5.8: FilterGroupAdapter Class Method Summary
The following is an example of how to use the PersistenceManager, Query, Filters and FilterGroups. Listing 5.17 shows the JavaBeans-compliant interface implemented by the class whose instance is to be persisted in the database.

Listing 5.17: interface to Persist

```java
public interface TestInterface {
    public int getNumber();
    public void setNumber(int value);
    public String getText();
    public void setText();
}
```

Listing 5.18 shows the DAO class that will be responsible for saving and retrieving the objects.

Listing 5.18: Data Access Object Class

```java
public class DAOClass {
    /**
     * Inject the framework PersistenceManager service
     */
    @Service
    private PersistenceManager pm;

    /**
     * Inject a default factory which returns the
     * auto-generated implementation of the TestInterface
     */
    @Service
    Factory<TestInterface> factory;

    /**
     * Method-scoped transactions to save/retrieve test
     * objects to/from the database.
     */
    @Transaction
    public void addObjects() {
        ...
    }

    @Transaction(readOnly = true)
    public List<TestInterface> getObjects() {
        ...
    }
}
```
In lines 6-7, the framework `PersistenceManager` service is injected into the `pm` field. In
lines 13-14, a default factory which returns instances of `TestInterface` is injected into the
factory field. The implementation of the `addObjects()` and `getObjects()` methods is
listed below.

```
Listing 5.19: DAOClass accessObjects() Method
1   @Transaction
2   public void addObjects()
3   {
4       for (int i = 0; i < 20; i++)
5           {
6               TestInterface obj = factory.newInstance();
7               obj.setNumber(i);
8               obj.setText(Long.toString(i));
9               pm.add(obj);
10           }
11   }
```

Listing 5.19 begins by creating a transactional boundary scoped to the `addObjects()` method. The `addObjects()` method then saves 20 objects to the database.

```
Listing 5.20: DAOClass getObjects() Method
1   @Transaction(readOnly = true)
2   public List<TestInterface> getObjects()
3   {
4       /**
5           * Framework Query service to retrieve TestInterface instances
6           */
7       @New
8       Query<TestInterface> query;
9
10       /**
11           * Create two filters to retrieve objects
12           */
13       Filter filter1 = new FilterAdapter("number", 3);
14       Filter filter2 = new FilterAdapter("number", FilterOperator.between , 5, 7);
15
16       query.addFilter(filter1);
17       TestInterface result = query.getResult();
18       printSummary(new TestInterface[]{result});
19       query.clearFilters();
20       query.addFilter(filter2);
21       Set<TestInterface> results = query.getResultSet();
22       printSummary(results.asList());
23
24       /**
25           * Combine all the filters into one query.
26   ```
Listing 5.20 begins by creating a read-only transaction scoped to the `getObjects()` method. In lines 7-8, a framework `Query service` is inserted into the variable `query`. On line 13, a `filter` to retrieve the object with the `number` field equal to 3 is instantiated. Line 14 is a `filter` to retrieve objects with the value of the `number` field between 5 and 7. The next two code blocks add the two `filters` in turn. They also invoke the `query` and print the results to standard output. On lines 28-34, the two `filters` are combined in a `FilterGroup` and added to the `query` object. The `query` is again executed and the results printed to standard output.

**Listing 5.21: Test Main Harness**

```java
public class Test {
    /**
     * This method will create a service context then
     * invoke the persistence operations.
     */
    @ServiceContext()
    public doPersistenceTest () {
        DAOClass daoObject = new DAOClass();
        daoObject.addObject ();
        daoObject.getObjects();
    }

    /**
     * Application main
     */
    public static main() {
        try {
            Test test = new Test();
            test.doPersistenceTest();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```
Listing 5.21 is the harness to launch the test application. In the `doPersistenceTest()` method, a service context is first created to make available the framework services. Then, an instance of the `DAOClass` is created to invoke the `addObject()` and `getObject()` methods. The test is invoked from the application `main()` method.
Chapter 6

Setup and Deployment

One of the benefits of the *i3Persistence Framework* is the ease with which it can be integrated into an application. This section outlines the steps necessary to set up and deploy the *i3Persistence Framework* in Eclipse.

6.1 Database: Setup and Connection

Connecting the *i3Persistence Framework* to a database is a simple two-step process:

First, the necessary database tables are created using the supplied *SerialSchema.sql* script. Figure 6.1 shows the location of the scripts in the *i3Datastore* project.

Currently, the *i3Persistence Framework* supports Sybase and MySQL for its backend RDBMS. The *i3Datasource* module uses canned SQL scripts to save and retrieve objects. These files are located in the /src/META-INF/sql directory of the *i3Datasource* project. The /src/META-INF/sql/common directory contains SQL scripts that are common between Sybase and MySQL. The

![](image)

Figure 6.1: *SerialSchema.sql* file location

/src/META-INF/sql/<database>/dml directories contain SQL scripts which are specific to a database.
Second, the `connection.properties` file must be configured with the JDBC database connection settings. Figure 6.2 shows the location of the META-INF directory containing the `connection.properties` file.

![Figure 6.2: connection.properties file location](image)

Each application using the i3Persistence Framework must have a `connection.properties` file in the META-INF directory at the root of where the compiled classes are stored. The `connection.properties` file contains name-value pair settings necessary for a JDBC driver to connect to a database. Figure 6.3 shows the contents of a `connection.properties` file configured to connect to a remote instance of a MySQL database.

![Figure 6.3: connection.properties file Settings](image)

The `pool-size` setting controls the number of connections in the connection-pool used to connect to the database. The `sql-resource-path` setting indicates the location of the SQL scripts specific to the target platform. The i3Persistence Framework ships with the jTDS (Sybase and MS SQL Server) and the MySQL native JDBC drivers and corresponding SQL scripts.
6.2 Preparing an Eclipse Project

Incorporating the *i3Persistence Framework* into an Eclipse project requires 1) the framework JAR files be added to the build and run environments and 2) the *i3Agent* be inserted into the VM via the Java Instrumentation mechanism.

The project *Java Build Path* settings determine the *compile-time classpath* for building an application in Eclipse. Figure 6.4 shows the JAR files included in the *Java Build Path* setting. These *compile-time* dependent JAR files correspond directly to the *Service Interface* modules outlined in Section 5.1.1. The remaining JAR files are used specifically by the *i3PersistenceTests* project as part of the testing framework.

The *runtime-classpath* is configured for each runtime configuration associated with a project. The JAR files necessary to run the application must be added to the *Classpath* setting in either a debug configuration or a runtime configuration. These *runtime* dependent JAR files correspond to the implementation modules outlined in Section 5.1.1. Figure 6.5 shows how the *Classpath* setting can be modified in an Eclipse project runtime configuration.
The final step is to enable the *i3Persistence Framework* Injection feature by hooking the *i3Agent* module into the Java VM at application bootup as shown in Figure 6.6.

A custom agent can be inserted into the VM’s bootup process by adding the `-javaagent` command-line option to the startup command. The syntax for the `-javaagent` option is:

```
java(w) -javaagent:jarpath[=options]
```
Chapter 7

Product Validation

7.1 Replicating the Modification Request Experiment

A project contributor usually modifies all the files necessary to accomplish a given task, then submits them together to the source repository in a single commit or Modification Request (MR) (see Figure 7.1, step 3). Knowing which files were modified at the same time allows researchers to analyze the logical coupling between files [15, 45].

Figure 7.1: Software Trails
Over the past decade, CVS has emerged as the preferred choice for source code repositories in many open source projects. Consequently, CVS repositories have long evolution periods (5-10 years) and represent a ready resource for research on software evolution [43]. Unfortunately, CVS does not record the list of files committed together in an MR. Instead, each file is committed as a separate transaction. This makes it difficult to determine which files were committed together.

In 2004, Zimmermann et al. proposed the fixed-time-window algorithm for rebuilding MRs [46, 18]. The following year, Zimmermann et al. proposed the sliding-time-window algorithm asserting it to be superior to the previous fixed-time-window algorithm [47].

To validate the sliding-time-window algorithm, German rebuilt the MRs for the Mozilla, Evolution, PostgreSQL and GNU gcc projects [18]. To validate the effectiveness of the i3Persistence Framework, we replicated the GNU gcc portion of German’s study first using the i3Persistence Framework and again using standard JDBC/SQL.

The following sections outline our replication process and compare the i3Persistence Framework and JDBC/SQL solutions.

7.1.1 The Sliding Time Window Algorithm

```java
// front(List) removes the front of the list
// top(list) and last(list) query the corresponding elements of the list
// Initialize the set of all MRs to empty
MRS = ∅
for each A in Authors do
    List = Revisions by A ordered by date
    do
        MR.list = (front(List))
        MR.sTime = time(MR.list)
        while first(List). time − MR.sTime ≤ δmax ∧
            first(List). time − last(MR.list). time ≤ τmax ∧
            first(List). log = last(MR.list). log ∧
            first(List). file ∈ MR.list do
            queue(MR.list, front(List))
        MRS = MRS ∪ (MR)
    do
    until List ≠ ∅
```

Figure 7.2: The sliding-time-window algorithm

German utilized a heuristic based on the sliding-time-window algorithm to rebuild the MRs. This algorithm takes two parameters as input: the maximum length of time that an MR can last δ_{max}, and the maximum distance in time between two file revisions τ_{max} (see figure 7.2) [18].
In the *sliding-time-window* algorithm, a file revision is included in a given MR if:

1. All the file revisions in the MR and the candidate file revision were created by the same author and have the same log message (a comment added by the developer during the commit).

2. The candidate file revision is at most $\tau_{max}$ seconds apart from at least one file revision in the MR.

3. The addition of the candidate file revision to the MR keeps the MR at most $\delta_{max}$ seconds long.

### 7.1.2 Assumptions

CVS maintains the details of file revisions in log files which can be accessed using the `cvs log` command. The command supports many options to customize the information returned from the source repository. Consequently, the accuracy of the MR reconstruction is directly related to the parameters used when invoking the `cvs log` command [13]. The command parameters that may affect the composition of the file revision information returned from the CVS repository are:

- **-d dates**
  
  Print information about revisions with a checkin date/time in the range given by the semicolon-separated list of dates.

- **-rrevision**
  
  Prints information about revisions given in the comma-separated list `revisions` of revisions and ranges.

**branch**

An argument that is a branch means all revisions on that branch.
Print information about revisions whose state attributes match one of the states given in the comma-separated list states.

The only information available from the original experiment about which parameters were used to retrieve the \texttt{cvs log} is:

“A snapshot of their CVS repositories was made on Feb 17, 2004 [18]. ”

Because the dataset and the \texttt{cvs log} command used in the original experiment were inaccessible, we proceeded with the replication experiment after making the following assumptions about the \texttt{cvs log} command used by German’s team:

- Include \texttt{-d"<2004-02-18"} to retrieve all revisions made before midnight of Feb 17, 2004
- Exclude \texttt{-r} to retrieve revisions for every branch
- Exclude \texttt{-s} to retrieve revision for all states

An additional assumption we made was that file revision information with empty commit comments were considered valid and included in the analysis of the log information.

7.1.3 Unraveling the CVS Log Format with \texttt{cvs2cl}

Although CVS revision history log information can be retrieved using the \texttt{cvs log} command, the format of the information is not fully documented. In the original experiment, German utilized a “clean room” approach to reverse-engineer the log format [18]. Since the time of German’s original experiment, many tools have emerged from the open source community that translate \texttt{cvs log} information into more human-readable formats.

Perl script \texttt{cvs2cl} is an industry standard tool that produces a GNU-style ChangeLog for CVS-controlled sources by running \texttt{cvs log} and parsing the output [13]. We modified \texttt{cvs2cl} to output the CVS file revision information. The \texttt{command-line} parameters we employed were as follows:
Listing 7.1: cvs2cl command-line options

```
cvs2cl -l "-d" "<2004-02-18"" --utc -f ChangeLog_2004-02-17.xml
```

The actual `cvs log` command generated by `cvs2cl` is `cvs log -d"2004-02-18"`, which complies with our assumptions of Section 7.1.2.

As part of our modification of `cvs2cl`, we wrapped user messages (comments) in `CDATA` tags to prevent any HTML markups contained in messages from confusing the `XML parser`. The `CDATA` tags also prevent parsing errors caused by invalid unsupported character encodings in messages.

7.1.4 The Application Design

The development plan for the replication experiment was to follow the phases outlined in `Bokeo`, namely extract, analyze, aggregate and present (see Section 1.2 for details). Because `Bokeo` is currently under development, we manually implemented each `Bokeo` phase.

7.1.4.1 Extraction Phase: The ChangelogExtractor Module

The `ChangelogExtractor` module encapsulates the `Extraction Phase` of `Bokeo`. Figure 7.3 is the UML class diagram and Table 7.1 is the detailed description of the classes that comprise the `ChangelogExtractor` module.

The `ChangelogExtractor` module is comprised of three class types: `XML Parsers`, `DTOs`, and `Service Interfaces`. The `XML Parsers` implement the `Service Interface` which is injected into classes in the `Analysis and Synthesis Phase` by the `i3Persistence Framework`. The `DTOs` are populated by the `XML Parsers` and passed to the clients via the `EventListener` `Service Interfaces`.

54
Due to the large size of the cvs2cl output file, it was not possible to use a DOM Parser to extract the revision information. Our solution to this problem was to use an XML Pull Parser (XPP) which requires less memory and performs better than DOM Parsers [36, 31].

Our first selection of an XPP technology was StAX2 [38]. Since the StAX2 technology was new to our team, we used the i3Persistence Framework Dependency Injection Pattern to decouple the parser implementation from the rest of the module. ChangelogStax2XmlParserImpl, our implementation of the StAX2 parser, was mapped for injection into the ChangelogParser Service Interface by registering it in the ServiceContext Initializer class, ChangelogParserInitializer.

Next, we implemented the DTOs to carry the file revision information to and from the database. We created the interfaces and utilized the i3Persistence auto-synthesis service to generate the implementations. Since the implementation auto-generation process is limited to only synthesizing property variables, complex data types such as List<> must be initialized with an instance of a List<>. The purpose of the ChangelogEntryDtoAbstractImpl and ChangelogEntryFileDtoAbstractImpl abstract classes is to lazy-load the List<>
### Table 7.1: ChangelogExtractor module class descriptions

<table>
<thead>
<tr>
<th>Class name</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangelogParser</td>
<td>Service Interface</td>
<td>Exposes the module’s functionality</td>
</tr>
<tr>
<td>ChangelogStax2ParserImpl</td>
<td>XPP</td>
<td>XML Pull Parser library based on the StAX2</td>
</tr>
<tr>
<td>ChangelogXPP3ParserImpl</td>
<td>XPP</td>
<td>XML Pull Parser library based on the XPP3</td>
</tr>
<tr>
<td>ChangelogParserEventListener</td>
<td>Event Listener Service Interface</td>
<td>Parser events listener interface</td>
</tr>
<tr>
<td>ChangelogParserInitializer</td>
<td>Injection Initializer Class</td>
<td>Registers service implementations for injection</td>
</tr>
<tr>
<td>ChangelogEntryDto</td>
<td>DTO Interface</td>
<td>Changelog Entry DTO interface</td>
</tr>
<tr>
<td>ChangelogEntryDtoAbstractImpl</td>
<td>Abstract DTO Class</td>
<td>ChangelogEntryDto partial implementation</td>
</tr>
<tr>
<td>ChangelogEntryFileDto</td>
<td>DTO Interface</td>
<td>Changelog Entry File DTO interface</td>
</tr>
<tr>
<td>ChangelogEntryFileDtoAbstractImpl</td>
<td>Abstract DTO Class</td>
<td>ChangelogEntryFileDto partial implementation</td>
</tr>
<tr>
<td>ChangelogTagDateDto</td>
<td>DTO Interface</td>
<td>Changelog TagDate DTO interface</td>
</tr>
<tr>
<td>DtoHelper</td>
<td>Utility Class</td>
<td>Converts XML Strings to Objects</td>
</tr>
<tr>
<td>ChangelogParserDtoInitializer</td>
<td>Injection Initializer Class</td>
<td>Registers service implementations for injection</td>
</tr>
</tbody>
</table>

The type properties of the ChangelogEntryDto and ChangelogEntryFileDto interfaces. The remainder of the interface getters/setter are automatically synthesized by the Injection Layer as described in Section 4.2.5. This i3Persistence Framework feature allowed us to rapidly implement the object infrastructure necessary to persist the CVS file revision information to the database. It also significantly reduced the amount of boiler-plate code necessary to implement the DTO interfaces.

One point to note about Figure 7.3 is the lack of dependencies between the Parser and Parser.Dto packages. Although the XML parsers in the Parser package use the DTOs in the Parser.Dto package, there is no direct association between the classes of the two packages. This is due to Dependency Injection where service objects are instantiated and injected into the consumer classes by the injection framework and not directly created by the consumer classes (see Appendix C).

The next step was to register the Initializer classes to be loaded at application startup. This was done by adding the Initializers to the injection-config.xml file in the META-INF directory.

The final step was to setup the database and configure the connection information in the connection.properties file.
Of the eight hours spent on implementing the `ChangelogExtractor` module, approximately seven hours were spent on tasks related to parsing the XML file with the remaining hour allocated to implementing the persistence logic and the DTOs.

### 7.1.4.2 Adapting to a Crisis

The initial test of the `ChangelogStax2XmlParserImpl` XML parser uncovered a flaw in the StAX2 XPP library: the implementation was not able to process the encoding format used in several of the file revision messages. Although the messages were embedded in `CDATA` tags, the StAX2 XPP library attempted to parse the messages.

An investigation of other XPP technologies revealed that the XPP3 Pull Parser library was capable of processing the XML file in its entirety. We implemented an XPP3-based XML Pull Parser and replaced the faulty StAX2-based implementation registration in the `ChangelogParserInitializer` file in approximately 30 minutes. Aside from implementing the new parser and modifying the injection registration, no other changes were necessary.

### 7.1.4.3 Analysis and Synthesis Phase: The ChangelogAnalyzer Module

The `ChangelogAnalyzer` module encapsulates the Analysis and Synthesis Phase, Aggregation Phase, and Presentation Phase of Bokeo. Figure 7.4 is the UML class diagram of the `ChangelogAnalyzer` module.

Table 7.2 is the detailed description of the classes that comprise the `ChangelogAnalyzer` module.
Rebuilding the CVS revision modification requests is a three-step process:

1. **Extraction Phase and Analysis and Synthesis Phase** – Invoke the `CvsChangelogLoaderApp` to parse the `cvs2cl` output XML file and store the file revision information in the database.

2. **Aggregation Phase** – Invoke `CvsChangelogModificationRequest` to generate the modification requests from the CVS file revision information.

3. **Presentation Phase** – Generate the report.

`CvsChangelogLoader` is the main worker class for the XML parsing process. To maximize portability, the **XML Parser** and database **Service Interface** implementations were
### Table 7.2: ChangelogAnalyzer module class descriptions

<table>
<thead>
<tr>
<th>Class name</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CvsChangelogLoaderApp</td>
<td>Application/Presentation</td>
<td>Main application to load the CVS logs</td>
</tr>
<tr>
<td>CvsChangelogLoader</td>
<td>Implementation</td>
<td>The implementation to load the CVS logs</td>
</tr>
<tr>
<td>CvsChangelogModificationRequestApp</td>
<td>Application/Presentation</td>
<td>Main application to rebuild the MRs</td>
</tr>
<tr>
<td>CvsChangelogModificationRequest</td>
<td>Implementation</td>
<td>The implementation to rebuild the MRs</td>
</tr>
<tr>
<td>CvsChangelogDB</td>
<td>Service Interface</td>
<td>The database Service Interface</td>
</tr>
<tr>
<td>i3PersistenceDB</td>
<td>Service Interface</td>
<td>The data access object Service Interface</td>
</tr>
<tr>
<td>JdbcDBImpl</td>
<td>Implementation Class</td>
<td>The i3Persistence database implementation</td>
</tr>
<tr>
<td>i3PersistenceDAOImpl</td>
<td>Implementation Class</td>
<td>The i3Persistence DAO implementation</td>
</tr>
<tr>
<td>JdbcDAOImpl</td>
<td>Implementation Class</td>
<td>The JDBC/SQL DAO implementation</td>
</tr>
<tr>
<td>CvsChangelogAnalyzerInitializer</td>
<td>Injection_INITIALIZER Class</td>
<td>Registers service implementations for injection</td>
</tr>
</tbody>
</table>

Injected into the CvsChangelogLoader by the framework. This allowed us to change the parser implementation between the StAX2 and XPP3 solutions by simply modifying the ChangelogParserInitializer Initializer class. We were also able to switch between the JDBC and i3Persistence based DB and DAO solutions by modifying the CvsChangelogAnalyzerInitializer class.

To compare the accuracy and productivity differences of using the i3Persistence Framework to technologies currently in use, we implemented a reference version of the CvsChangelogDB and CvsChangelogDAO interfaces using JDBC and SQL. The ChangelogParser injection implementation is defined in the ChangelogExtractor module injection-config.properties file (see Appendix D), and the CvsChangelogDB and CvsChangelogDAO injection implementations are defined in the ChangelogAnalyzer module injection-config.properties file (see Appendix E).

The MR statistics of Table 7.3 were collected by a custom ChangelogEventListener passed to the ChangelogParser by the CvsChangelogModificationRequest class. The data for Figures 7.5 and 7.6 were collected by CvsChangelogDAO during the MR rebuilding process.
7.2 Validating the Implementation

Because German’s original study provides little information on implementation details against which to compare our application, we validated our implementation against the *sliding-time-window* algorithm [18, 47]. This algorithm defines four basic steps (see Figure 7.2):

1. Retrieve all distinct file revision authors.
2. Retrieve all distinct commit messages by each author.
3. Retrieve all files revisions having the same commit message by the same author.
4. Rebuild the MRs using the files retrieved in (3)

Steps (1) to (3) are implemented in the DAO classes. Step (4), the business module for the Modification Request Rebuilder, is implemented in the `CvsChangelogModificationRequest` class. We used the JUnit tests detailed in Appendix F to validate our implementation.

7.3 Replication Results

Table 7.3 shows the main statistics of the GNU gcc projects for the original and the replicated experiments.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Authors</th>
<th>Files</th>
<th>Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>214</td>
<td>24,463</td>
<td>60,311</td>
</tr>
<tr>
<td>Replicated</td>
<td>214</td>
<td>22,875</td>
<td>54,647</td>
</tr>
<tr>
<td>Error %</td>
<td>0.0%</td>
<td>-6.49%</td>
<td>-9.06%</td>
</tr>
</tbody>
</table>

Table 7.3: CVS statistics for the GNU gcc project

Figure 7.5 shows the distribution of the number of files in an MR (normalized to values from 0 to 1). Figure 7.6 shows the MRs per month during 2003.

Figure 7.6 shows that approximately 23 of 26 data points of the replicated experiment are within 1.0% of the original experimental results (see Appendix I.1). Figure 7.6 shows
that the replicated data has basically the same progression as the original data except for July when the replicated data transitions from a negative to a positive offset (see Appendix J.1).

### 7.4 Conclusion

The differences between the original and replicated experiment results can be attributed to one or more, but not limited to, the following factors:

- **Implementation technologies** – We used Java while the original experiment used Perl.
- **Database technologies** - The *i3Persistence Framework* is a potential source of data variance.
- **Datasets** – The output of *cvs2pl* may not match the dataset used in the original experiment.
- **Algorithms** – We could have misinterpreted the *sliding-time-window* algorithm of the original experiment.

The following subsections discuss our analysis of each of the aforementioned factors.
7.4.1 Implementation Technologies

The differences between Java and Perl which may affect our experiment results are 1) the date/time functions 2) the math library and 3) text handling. In our replicated experiment application implementation, we used the industry-accepted JodaTime libraries to convert text representations of date/time to DateTime objects. We assume that the original experiment used the Perl gmtime() and localtime() libraries which are implemented in C. The Java and Perl functions are equivalent in functionality and should not cause any discrepancies in the data.

The math functions used are basic addition and subtraction of Long Integers. Java and Perl math function produce the same outputs.

Text handling is one area of difference between Java and Perl that may appreciably influence experiment results. This is due to the various text encoding formats used by project contributors. Text in certain native encodings can be construed by XML Parsers as invalid characters. To ensure that file revision messages are compared in its original form, we embedded message text in CDATA tags. When XML parsers encounter the CDATA tag, the enclosed text is passed directly to the invoking application without any processing, thus bypassing any potential encoding errors.

7.4.2 Database Technologies

Our JUnit tests proved that the i3Persistence Framework-based implementations functioned as designed (see Appendix F). Also, the comparison of the replicated experimental results using three independent database solutions (the i3Persistence Framework, JDBC/SQL and Berkeley DB technologies) was identical (see Appendix H). This consistency strengthens our argument that differences in database technologies are not a factor in the disparity between the original and replicated experimental results.
7.4.3 Datasets

Without the original dataset, we cannot quantitatively discuss the differences introduced by the dataset used in the replicated experiment. But because cvs2cl is an industry-accepted technology shipped with most Linux distributions, we feel confident that if any variances exist in the datasets, the cvs2cl-based output is more accurate than that obtained by German’s reversed-engineered solution.

7.4.4 Algorithms

Without the original implementation details, we are not able to assess the differences in the approaches used in the original and the replicated solutions. Although we can estimate the potential margin of error between the original and replicated implementations based on the relative simplicity of the algorithm, a definitive comparison is not possible until such time when a more quantitative analysis is conducted.

7.5 Comparison of Technologies

Tables 7.4 and 7.5 compare the i3Persistence Framework and JDBC/SQL solutions based on three productivity factors: development time, lines of code, and McCabe Cyclomatic Complexity values [35].

<table>
<thead>
<tr>
<th>DB Class</th>
<th>Development Time</th>
<th>Lines of Code</th>
<th>McCabe Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>≈ 3 hours</td>
<td>84</td>
<td>5</td>
</tr>
<tr>
<td>i3Persistence</td>
<td>≈ 15 min.</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>Ratio (JDBC:i3)</td>
<td>12:1</td>
<td>3.8:1</td>
<td>2.5:1</td>
</tr>
</tbody>
</table>

Table 7.4: JDBC vs. i3Persistence: Database class

Our experience showed that we were able to develop the DAO and DTO logic approximately 10 times faster using the i3Persistence Framework than using JDBC/SQL. Furthermore, the i3Persistence Framework based DB and DAO modules required fewer lines of code.
and had, in the case of the DAO classes, significantly lower McCabe’s Cyclomatic Complexity values [29]. The majority of the time in implementing the JDBC/SQL versions of the DAO and DTO modules was spent:

1. Optimizing the schema.
2. Setting up the SQL query.
3. Mapping the JDBC table representations and data types to our object model.

In comparison, the *i3Persistence Framework*:

1. Has a fixed schema provided in the project that needs to be run once per database.
2. Requires no SQL.
3. Supports the persisting of objects, which allowed us to work in the object-oriented paradigm and with native Java data types.

The ease of maintenance of the *i3Persistence Framework* solutions, as reflected by the low McCabe’s Cyclomatic Complexity values, was made evident in the relative ease with which we replaced the faulty StAX2 version of the XPP with the XPP3 version (see Section 7.1.4.2). The decoupling nature of *i3Persistence Framework* DI feature localized the system integration changes to a single configuration file. The injection feature also allowed us to easily switch between the JDBC/SQL and *i3Persistence Framework* versions of the DAO and DTO modules.

The research portion of the replication experiment, investigating the cause of the differences between the original and replicated results, consumed the majority of the four week replication project period. During this time, various permutations of the experimental

<table>
<thead>
<tr>
<th>DAO Class</th>
<th>Development Time</th>
<th>Lines of Code</th>
<th>McCabe Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>≈ 5 hours</td>
<td>226</td>
<td>9</td>
</tr>
<tr>
<td>i3Persistence</td>
<td>≈ 30 min</td>
<td>130</td>
<td>2</td>
</tr>
<tr>
<td>Ratio (JDBC:i3)</td>
<td>10:1</td>
<td>1.7:1</td>
<td>4.5:1</td>
</tr>
</tbody>
</table>

Table 7.5: JDBC vs. *i3Persistence*: DAO class
variables were employed in an attempt to force a convergence of the original and replicated experiment results. Although we were unsuccessful in obtaining consistency in the data, modifying the JDBC/SQL solution required approximately five times more resources when compared to the resource demands of the *i3Persistence Framework* solution.

One noticeable overhead of using the *i3Persistence Framework* was the up-front time investment necessary for developers to learn to use the technology. This “learning curve” was evident in the differences in productivity between developers who were familiar with the framework and those who were working with it for the first time. In general, we observed that developers attained proficiency with the *i3Persistence Framework* relatively quickly when compared to developers working with JDBC/SQL. We attribute this to 1) the fact that the *i3Persistence Framework* is based on familiar standards and technologies [25, 26, 27] and 2) the relative simplicity of the *i3Persistence Framework* APIs when compared to the complex APIs and language syntax of JDBC/SQL.
Chapter 8

Conclusion

The Sequoia Lab at BYU is actively developing an extensible framework, Bokeo, to help researchers conduct artifact-based empirical experiments on software evolution. Researchers can use Bokeo to chain existing components and custom plug-ins into workflows to extract, analyze and visualize derived metrics and data from software repositories. The i3Persistence Framework was developed to allow researchers to 1) easily incorporate data-persistence features in their Bokeo plug-ins and 2) to promote developer productivity.

Although Bokeo is a general purpose framework for conducting artifact-based empirical experiments on software evolution, this industrial thesis focuses on how the i3Persistence Framework facilitates the replication of artifact-based empirical experiments on data from software repositories. The i3Persistence Framework promotes a Java-centric development model which allows researchers to employ rapid write-debug-modify cycles when implementing data-centric tools for their experimental replication efforts. DI, annotation-based feature invocation, auto-implementation synthesis, and simplified persistence APIs 1) reduce the need to write boilerplate code and 2) abstract technology and complexity behind easy to use interfaces. The result is that researchers are able to focus on the “business” aspects of their replication processes.

The initial release of the i3Persistence Framework focused primarily on features that would increase developer productivity. Our experience showed that the learning curve experienced by developers new to the framework was relatively small when compared to that of using JDBC/SQL. Once developers gained proficiency with our new technology, they were
able to accomplish implementation tasks quickly and with relatively few lines of code. We attribute this increase in productivity to the fact that the *i3Persistence Framework* utilizes existing standards (annotations, object-oriented development, the Java programming language), de facto productivity design patterns (DI and Service Lookup), and a simple, terse API set.

Although our initial motivation to develop a new persistence framework was to compensate for the lack of existing tools which would facilitate the plug-in development requirements of *Bokeo*, the *i3Persistence Framework* was not designed for this purpose alone. We designed the *i3Persistence Framework* to address the general need in the software artifact-based empirical research community to increase developer productivity and decrease the cost of software maintenance.

We realize that there is yet much work to do to ready the *i3Persistence Framework* for general consumption. Security, scalability, performance, etc. are just a few of the areas yet to be addressed. Since the start of this industrial thesis, many tools have been developed that address these concerns, but we have yet to find any tool that combines the level of API simplicity and productivity features as does the *i3Persistence Framework*. It is our hope that our work will serve to further the development of tools not only in the area of experimental replication, but in other areas of data-centric research.
Chapter 9

Future Work

9.1 Point-in-Time Architecture

One of the fundamental requirements when replicating experiments on data from software repositories is the need for reference datasets along various points of the original experiment’s computation process. These reference datasets can be used to validate the replication process. One way to gather reference data is to use a Point-in-Time Architecture (PTA) database.

A Point-in-Time Architecture system is able to present an image of the database as it existed at any previous point in time, without destroying the current image. In effect, a PTA maintains a searchable audit trail of all data that is persisted in the database over the life of that database.

9.2 Optimize Speed

A known trade-off when designing the database schema for the i3Persistence Framework is its performance limitations. The highly-normalized nature of the i3Persistence database tables requires multiple joins when storing and retrieving objects. Additionally, current RDBMSs distribute the computational load onto the database server while the i3Persistence Framework localizes it on the application server.

Performance issues were ignored in the initial release of the i3Persistence Framework with the justification that advancements in hardware performance will compensate for most of the performance issues and that the majority of the expense when developing database applications is in manpower costs versus hardware costs.
9.3 Support Large Datasets

Currently, the *i3Persistence Framework* retrieves the entire object hierarchy in memory for each query. This strategy is impractical for large datasets. Memory usage can greatly be reduced via hot loading and unloading of business objects.

9.4 Object Version Migration

Currently, the *i3Persistence Framework* does not track changes to class structure. This feature is critical in a prototyping environment such as when conducting one-off research experiments. One solution is to version object instances and use *Javassist* to generate version-specific class structures at runtime.
Bibliography


Appendix A

Separating Responsibilities with a Component-Layered Architecture

Software architecture is commonly articulated in the form of the major components and services of the system and how they interact. A software component can be defined as a grouping of software that’s intended to be used, without change, by an application that is outside the scope of control or knowledge of the writers of the component. When using the component, application source code is not changed, although the component’s behavior may be altered by extending it in ways allowed by the component writers.

The component-layered approach is one of the most common techniques used by software designers to organize a complicated software system. In this scheme, each layer is composed of one or more components that are tightly coupled. This complexity is abstracted from the outside world by the Service Interface. The layers are organized so that higher layers use various services defined by lower layers, but the lower layer is unaware of the higher layer. Furthermore, each lower layer usually hides its lower layers from the higher layers.

In the component-layered architecture depicted in Figure A.1, layer N relies on services from layer N-1 to provide a service to layer N+1 and N+2. In each layer, interfaces define the services offered.

Breaking down a system into layers has a number of important benefits:

- Complexity abstraction – The implementation details of a layer are hidden by the Service Interface, promoting simpler interactions between components and layers.
• Separation of Concerns – It is possible to understand a single layer as a coherent whole without knowing much about the other layers.

• Modularization – Since the service required from a lower layer is independent of its implementation, alternative implementations of the same basic services can be used without adversely affecting the system.

• Reduce Coupling – Dependency is isolated to adjacent layers which minimizes cross-layer coupling.

• Standardization – *Service Interface* present opportunities to standardize layer interaction.

• Code Reuse – As depicted in Figure A.1, a component can be reused in multiple layers and a single layer can service multiple clients, which promotes code reuse.
Appendix B

Decoupling Modules with the Dependency Inversion Principle

The *Service Interface* depicted in Figure 5.1 is an implementation of the *Dependency Inversion Principle*, which serves to resolve the shortcomings of the standard component-layered architecture where higher-level layers are coupled to lower-level layers. The primary disadvantages of the standard component-layered architecture are that designs tend to be:

- Rigid (hard to change due to dependencies, especially since dependencies are transitive);
- Fragile (changes cause unexpected bugs);
- Immobile (difficult to reuse across applications due to implicit dependence on current application code).

Robert Martin formulated the *Dependency Inversion Principle* (DIP) to address the disadvantages of the standard Component-Layered Architecture (see Appendix 4.2.1) [33]. The DIP states:

1. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

The goal of the DIP is to decouple higher-level components from their dependency upon lower-level components. This may be achieved by creating *interfaces*, which define the component’s external needs, as part of the higher-level component package. This
allows the component to be isolated from any particular implementation of the provided interface, thus increasing the component’s portability. In Java, this decoupling is achieved through the use of interfaces. Figure B.1 illustrates this relationship.

The i3Persistence Framework achieves reduction in coupling by using a Dependency Inversion-based Component-Layered Architecture (Section A) and Dependency Injection to “glue” the components and layers together into a cohesive software solution (Section C).

This decoupling facilitates the anonymous distributed development of the i3Persistence Framework components and layers.
Appendix C

Connecting Components and Layers with the Dependency Injection Pattern

A common issue facing developers of application frameworks is how to wire together the different elements of the framework. The complexity of this issue is magnified when portions of the software are built by different teams with little knowledge of each other. The i3Persistence Framework supports the Dependency Injection Pattern (DI) to loosely couple elements of the framework to the client application.

DI, a specific form of Inversion of Control, is an object-oriented design pattern whose purpose is to reduce coupling between software components by separating behavior from dependency resolution [14]. In most software applications, consumer components are tightly coupled to service components because of the need for the consumer component to explicitly state all the details of the creation of an instance of the service component. Figure C.1 depicts the hard coupling between consumer and service components.

![Figure C.1: Consumer-Service Dependency](image)

DI eliminates the need for explicit service instantiation by externally injecting a reference to the service component inside the consumer component. In other words, objects are configured by an external entity. DI is an alternative to having the object configure itself.

In Figure C.2, the DI Container first resolves all of the server component’s dependencies, meaning all the dependency injection requirements of the server component itself.
The DI Container then injects the fully resolved reference to the service component into the consumer component. For example, if `ServiceComponentImpl` is dependent on class A, the DI Container will execute the following tasks when the ConsumerComponent is instantiated:

1. Instantiate `ServiceComponentImpl`
2. Instantiate class A
3. Inject object A into the instance of `ServiceComponentImpl`
4. Inject `ServiceComponentImpl` into the instance of `ConsumerComponent`

Figure C.3 depicts the resulting dependencies.
The DI design pattern removes the hard dependency of the \texttt{ConsumerComponent} on the \texttt{ServiceComponentImpl} by introducing an external entity, the DI Container. The DI container, in turn, has dependencies to the \texttt{ServerComponentImpl} and the \texttt{ConsumerComponent}. The \textit{i3Persistence Framework} eliminates the hard injection dependency of the DI Container to the ConsumerComponent by using Javassist and byte-code manipulation (see Section C.3 for details).
Appendix D

ChangelogExtractor Module injection-config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE injection-config PUBLIC "-//Sequoia Lab//DTD Injection Configuration//EN" "/META-INF/dtd/injection-config.dtd">

<injection-config>
  <injection-class-list>
    <injection-class regex="edu\.byu\.cs\.sequoia\.bokeo\..*+"/>
  </injection-class-list>
  <initialization-class-list>
    <initialization-class classname="edu.byu.cs.sequoia.bokeo.changelogextractor.parser.ChangelogParserInitializer"/>
    <initialization-class classname="edu.byu.cs.sequoia.bokeo.changelogextractor.parser.dto.ChangelogParserDtoInitializer"/>
  </initialization-class-list>
</injection-config>
```
Appendix E

ChangelogAnalyzer Module injection-config.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE injection-config PUBLIC "-//Sequoia Lab//DTD Injection Configuration//EN" "/META-INF/dtd/injection-config.dtd">
3  
4  <injection-config>
5    <injection-class-list>
6      <injection-class regex="edu\.byu\.cs\.sequoia\.bokeo\..\..\.*"/>
7    </injection-class-list>
8    <initialization-class-list>
9      <initialization-class classname="edu.byu.cs.sequoia.bokeo.changelog analyzer.app.db.CvsChangelogAnalyzerInitializer"/>
10     </initialization-class-list>
11  </injection-config>
```
Appendix F

ChangelogDAOITest.java

```java
package edu.byu.cs.sequoia.bokeo.changeloganalyzer.app.db;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.Date;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

import junit.framework.Assert;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

import edu.byu.cs.sequoia.bokeo.changeloganalyzer.app.
ModificationRequestBuilder;
import edu.byu.cs.sequoia.bokeo.changeloganalyzer.app.db.ChangelogDAO;
import edu.byu.cs.sequoia.bokeo.changeloganalyzer.app.db.dto.
ModificationRequestDto;
import edu.byu.cs.sequoia.bokeo.changeloganalyzer.app.db.dto.
CvsLogFileDto;
import gov.utah.ush.i3.annotations.Service;
```
import gov.utah.ush.i3.annotations.ServiceContext;
import gov.utah.ush.i3.annotations.ServiceContextType;
import gov.utah.ush.i3.service.Factory;

public class ChangelogDAOTest {
    @Service
    ChangelogDAO dao;

    @BeforeClass
    @ServiceContext(contextType = ServiceContextType.
    containerContextInitializer)
    public static void setup()
    {
        // Scope the service context to the class;
    }

    @AfterClass
    @ServiceContext(contextType = ServiceContextType.
    containerContextFinalizer)
    public static void teardown()
    {
        // Clean-up resources
        dao.close();
    }

    /**
     * Test the API to retrieve all the revision files.
     */
    @Test
    public void testGetFiles()
    {
        new Object()
        {
            Long fileCount = 0L;
        }
    }
void run()
{
    // Check each file
    Iterator<Object> itr = dao.getFiles();
    while (itr.hasNext())
    {
        fileCount++;
    }
    Assert.assertEquals(442075, fileCount.longValue());
}
}.run();
}

/**
 * Test the API which returns all the authors.
 */
@Test
public void testGetAuthors()
{
    final Set<String> authors = new HashSet<String>();
    new Object()
    {
        void run()
        {
            Iterator<String> itr = dao.getAuthors();
            while (itr.hasNext())
            {
                authors.add(itr.next());
            }
        }.run();
        Assert.assertEquals(214, authors.size());
    }
}
* Test API to return messages by authors.

```java
@Test
public void testGetMessagesByAuthor()
{
    final Set<String> missingMsgs = new HashSet<String>();
    new Object()
    {
        void run()
        {
            // Check each author
            Iterator<String> itr = dao.getAuthors();
            while (itr.hasNext())
            {
                // Manually compile the messages per author
                Set<String> msgs = new HashSet<String>();
                String author = itr.next();
                Iterator<Object> filesByAuthor = dao.getFilesByAuthor(author);
                while (filesByAuthor.hasNext())
                {
                    CvsFile file = (CvsFile) filesByAuthor.next();
                    msgs.add(file.getMsg());
                }
                // Get the messages using the unique index
                Iterator<String> msgItr = dao.getMessagesByAuthor(author);
                while (msgItr.hasNext())
                {
                    String msg = msgItr.next();
                    if (!msgs.contains(msg))
                    {
                        missingMsgs.add(msg);
                    }
                }
            }
        }
    }
}
```

// Remove the message so that duplicates are considered as
```java
137         errors
138         msgs.remove(msg);
139     }
140 }
141 }
142 }.run();
143     Assert.assertEquals(0, missingMsgs.size());
144 }
145
146 /**
147 * Test the API which returns the files with the same
148 * message and author.
149 */
150 @Test
151 public void testGetFilesByAuthorAndMessage()
152 {
153     new Object()
154     {
155         void run()
156         {
157             // Check each author
158             Iterator<String> itr = dao.getAuthors();
159             while (itr.hasNext())
160             {
161                 // Manually compile the messages per author
162                 Map<String, List<CvsFile>> filesByMsg = new HashMap<String,
163                                                 List<CvsFile>>();
164                 String author = itr.next();
165                 Iterator<Object> filesByAuthor = dao.getFilesByAuthor(author);
166                 while (filesByAuthor.hasNext())
167                 {
168                     CvsFile file = (CvsFile) filesByAuthor.next();
169                     if (!filesByMsg.containsKey(file.getMsg()))
170                         {
```
filesByMsg.put(file.getMsg(), new ArrayList<CvsFile>());
}
filesByMsg.get(file.getMsg()).add(file);

// Get the files for each generated message and author
for (String msg : filesByMsg.keySet())
{
    List<CvsFile> generatedFiles = filesByMsg.get(msg);

    // Get the indexed files
    Iterator<Object> fileItr = dao.getFilesByAuthorAndMessage(
            author, msg);

    // Compare the generated and indexed files
    while (fileItr.hasNext())
    {
        boolean found = false;
        CvsFile indexedFile = (CvsFile) fileItr.next();
        for (int i = 0; i < generatedFiles.size(); i++)
        {
            CvsFile generatedFile = generatedFiles.get(i);
            if (fileEquals(indexedFile, generatedFile))
            {
                generatedFiles.remove(i);
                found = true;
                break;
            }
        }
    }

    if (!found)
    {
        Assert.fail("Indexed file not found in generated.");
    }
}
if (generatedFiles.size() > 0) {
    Assert.fail("Generated had more files then indexed.");
}
}

/**
 * Test the MR rebuilding logic.
 */
@Test
public void testGetMRs() {
    new Object() {
        void run() {
            // Create the MRs
            List<ModificationRequestDto> mrs = new ArrayList<ModificationRequestDto>();
            generateMRs(mrs);
            Assert.assertEquals(4, mrs.size());
        }
    }.run();
}

/***************************************************************
 * Utility functions.
 ************************************************************/

boolean fileEquals(CvsFile file1, CvsFile file2) {
    Assert.assertEquals(file1.getId(), file2.getId());
    Assert.assertEquals(file1.getAuthor(), file2.getAuthor());
}
Assert.assertEquals(file1.getCvsstate(), file2.getCvsstate());
Assert.assertEquals(file1.getMsg(), file2.getMsg());
Assert.assertEquals(file1.getTime(), file2.getTime());
Assert.assertEquals(file1.getRevision(), file2.getRevision());

return true;

/**
 * Generate test file revision data.
 */
void generateMRs(final List<ModificationRequestDto> mrs)
{
    new Object()
    {
        @Service
        Factory<CvsLogFileDto> fileFactory;

        public void run()
        {
            PriorityQueue<CvsFile> files = new PriorityQueue<CvsFile>(1,
                new CompareCvsFileByTime());
            Date now = new Date();
            long start = now.getTime();

            // First MR has 5 files
            files.add(makeFile(1L, "test", "Test message", start + 2));
            files.add(makeFile(2L, "test", "Test message", start + 4));
            files.add(makeFile(3L, "test", "Test message", start + 6));
            files.add(makeFile(4L, "test", "Test message", start + 8));
            files.add(makeFile(5L, "test", "Test message", start + 10));

            // Second MR has 3 files
            start += 650000;
            files.add(makeFile(6L, "test", "Test message", start + 2));
        }
    }
}
files.add(makeFile(7L, "test", "Test message", start + 4));
files.add(makeFile(8L, "test", "Test message", start + 6));

// Third MR has 2 files
start += 650000;
files.add(makeFile(9L, "test", "Test message", start + 2));
files.add(makeFile(10L, "test", "Test message", start + 4));

// Fourth MR has 2 files
start += 650000;
files.add(makeFile(11L, "test4", "Test message", start));

ModificationRequestBuilder builder = new ModificationRequestBuilder();
mrs.addAll(builder.build(files, 450000L, 6000000L));
Assert.assertEquals(4, mrs.size());
}

CvsLogFileDto makeFile(Long id, String author, String msg, long time)
{
    CvsLogFileDto f = fileFactory.newInstance();
f.setId(id);
f.setAuthor(author);
f.setMsg(msg);
f.setTime(time);
    return f;
}

/**
 * Comparator to sort files by descending time.
 */
class CompareCvsFileByTime implements Comparator<CvsFile>
{
    @Override
    public int compare(CvsFile o1, CvsFile o2)
{ return o1.getTime().compareTo(o2.getTime()); }

}.run();

}.run();

}
Appendix G

ModificationRequestBuilder.java

```java
package edu.byu.cs.sequoia.bokeo.changeloganalyzer.app;

import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

import edu.byu.cs.sequoia.bokeo.changeloganalyzer.app.db.bdb.ModificationRequestDto;
import edu.byu.cs.sequoia.bokeo.changelogextractor.parser.dto.CvsFile;

/**
 * Implementation of the sliding-time-window algorithm.
 *
 * @author scott
 *
 */

public class ModificationRequestBuilder {

    /**
     * Build the modification requests.
     *
     *
     * @param files
     * @param distanceMax
     * @param periodMax
     * @return
     */

    public List<ModificationRequestDto> build(PriorityQueue<CvsFile> files, )
```
List<ModificationRequestDto> mrs = new ArrayList<>()

// Bootstrap the process:
// 1) Create a new MR
// 2) Assign the first revision
// 3) Set sTime (start time) and pTime (previous time)
// to the first revision time.
ModificationRequestDto mr = new ModificationRequestDto();
CvsFile currentFile = files.poll();
mr.getFiles().add(currentFile.getId());
mr.setTime(currentFile.getTime());
Long startTime = currentFile.getTime();
Long previousTime = startTime;

// Check each file for distance and period
while (files.size() > 0)
{
    // Get the current file time
    currentFile = files.poll();
    Long currentTime = currentFile.getTime();

    // Check the distance and period
    if ((currentTime - previousTime <= distanceMax) && (currentTime - )
        startTIme <= periodMax))
    {
        // Add the revision to the MR
        mr.getFiles().add(currentFile.getId());
        previousTime = currentTime;
    }
    else
    {

// Save the old MR and start a new MR
mrs.add(mr);

// Create the next MR
mr = new ModificationRequestDto();

// Bootstrap the process again.
mr.getFiles().add(currentFile.getId());
mr.setTime(currentFile.getTime());
startTime = previousTime = currentFile.getTime();
}
}

// Save the last one
if (mr.getFiles().size() > 0)
{
mrs.add(mr);
}

return mrs;
Appendix H

*i3Persistence Framework*, JDBC/SQL and Berkeley DB Test Results

<table>
<thead>
<tr>
<th>Database</th>
<th>Authors</th>
<th>Files</th>
<th>Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>i3Persistence (MySQL)</td>
<td>214</td>
<td>22,875</td>
<td>54,647</td>
</tr>
<tr>
<td>JDBC/SQL (MySQL)</td>
<td>214</td>
<td>22,875</td>
<td>54,647</td>
</tr>
<tr>
<td>Berkeley DB</td>
<td>214</td>
<td>22,875</td>
<td>54,647</td>
</tr>
</tbody>
</table>

Table H.1: Comparison of GNU gcc cvs log Analysis
Appendix I

Gnu gcc: Distribution of the Number of Files in an MR

<table>
<thead>
<tr>
<th># Files</th>
<th>Replicated</th>
<th>Original</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1744</td>
<td>0.1570</td>
<td>1.74%</td>
</tr>
<tr>
<td>2</td>
<td>0.4213</td>
<td>0.4240</td>
<td>-0.27%</td>
</tr>
<tr>
<td>3</td>
<td>0.1188</td>
<td>0.1180</td>
<td>0.08%</td>
</tr>
<tr>
<td>4</td>
<td>0.0946</td>
<td>0.0800</td>
<td>1.46%</td>
</tr>
<tr>
<td>5</td>
<td>0.0428</td>
<td>0.0600</td>
<td>-1.72%</td>
</tr>
<tr>
<td>6</td>
<td>0.0326</td>
<td>0.0410</td>
<td>-0.84%</td>
</tr>
<tr>
<td>7</td>
<td>0.0197</td>
<td>0.0250</td>
<td>-0.53%</td>
</tr>
<tr>
<td>8</td>
<td>0.0146</td>
<td>0.0150</td>
<td>-0.04%</td>
</tr>
<tr>
<td>9</td>
<td>0.0101</td>
<td>0.0110</td>
<td>-0.09%</td>
</tr>
<tr>
<td>10</td>
<td>0.0086</td>
<td>0.0090</td>
<td>-0.04%</td>
</tr>
<tr>
<td>11</td>
<td>0.0064</td>
<td>0.0070</td>
<td>-0.06%</td>
</tr>
<tr>
<td>12</td>
<td>0.0059</td>
<td>0.0060</td>
<td>-0.01%</td>
</tr>
<tr>
<td>13</td>
<td>0.0045</td>
<td>0.0050</td>
<td>-0.05%</td>
</tr>
<tr>
<td>14</td>
<td>0.0037</td>
<td>0.0040</td>
<td>-0.03%</td>
</tr>
<tr>
<td>15</td>
<td>0.0046</td>
<td>0.0035</td>
<td>0.11%</td>
</tr>
<tr>
<td>16</td>
<td>0.0033</td>
<td>0.0030</td>
<td>0.03%</td>
</tr>
<tr>
<td>17</td>
<td>0.0043</td>
<td>0.0025</td>
<td>0.18%</td>
</tr>
<tr>
<td>18</td>
<td>0.0017</td>
<td>0.0020</td>
<td>-0.03%</td>
</tr>
<tr>
<td>19</td>
<td>0.0019</td>
<td>0.0020</td>
<td>-0.01%</td>
</tr>
<tr>
<td>20</td>
<td>0.0014</td>
<td>0.0015</td>
<td>-0.01%</td>
</tr>
<tr>
<td>21</td>
<td>0.0012</td>
<td>0.0010</td>
<td>0.02%</td>
</tr>
<tr>
<td>22</td>
<td>0.0012</td>
<td>0.0010</td>
<td>0.02%</td>
</tr>
<tr>
<td>23</td>
<td>0.0012</td>
<td>0.0010</td>
<td>0.02%</td>
</tr>
<tr>
<td>24</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.00%</td>
</tr>
<tr>
<td>25</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Table I.1: Distribution of the Number of Files in an MR
Appendix J

Gnu gcc: Modification Requests per Month, 2003

<table>
<thead>
<tr>
<th>Month</th>
<th>Replicated</th>
<th>Original</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1198</td>
<td>1460</td>
<td>-21.87%</td>
</tr>
<tr>
<td>2</td>
<td>1164</td>
<td>1220</td>
<td>-4.81%</td>
</tr>
<tr>
<td>3</td>
<td>1233</td>
<td>1486</td>
<td>-20.52%</td>
</tr>
<tr>
<td>4</td>
<td>967</td>
<td>1157</td>
<td>-19.65%</td>
</tr>
<tr>
<td>5</td>
<td>890</td>
<td>984</td>
<td>-10.56%</td>
</tr>
<tr>
<td>6</td>
<td>1376</td>
<td>1520</td>
<td>-10.47%</td>
</tr>
<tr>
<td>7</td>
<td>1666</td>
<td>1380</td>
<td>17.17%</td>
</tr>
<tr>
<td>8</td>
<td>875</td>
<td>935</td>
<td>-6.86%</td>
</tr>
<tr>
<td>9</td>
<td>888</td>
<td>957</td>
<td>-7.77%</td>
</tr>
<tr>
<td>10</td>
<td>1184</td>
<td>1250</td>
<td>-5.57%</td>
</tr>
<tr>
<td>11</td>
<td>893</td>
<td>911</td>
<td>-2.02%</td>
</tr>
<tr>
<td>12</td>
<td>1016</td>
<td>1097</td>
<td>-7.97%</td>
</tr>
</tbody>
</table>

Table J.1: Modification Requests Per Month, 2003
Appendix K

UML Diagrams

K.1 i3Datasource: Module Statistics and Package Diagram

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Packages</td>
<td>7</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>62</td>
</tr>
<tr>
<td>Number of Interfaces</td>
<td>6</td>
</tr>
<tr>
<td>Total Lines of Code</td>
<td>4011</td>
</tr>
</tbody>
</table>

Table K.1: i3Datasource Module Metrics

Figure K.1: i3Datasource Module Package Diagram
K.2  i3InjectionApi: Module Statistics and Class Diagram

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Packages</td>
<td>5</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>6</td>
</tr>
<tr>
<td>Number of Interfaces</td>
<td>20</td>
</tr>
<tr>
<td>Total Lines of Code</td>
<td>322</td>
</tr>
</tbody>
</table>

Table K.2: i3InjectionApi Module Metrics

Figure K.2: i3InjectionApi Module Class Diagram
K.2.1 i3InjectionApi: Service Module Class Diagram

Figure K.3: i3InjectionApi Service Module Class Diagram
K.2.2 i3InjectionApi: Modules Class Diagram

Figure K.4: i3InjectionApi Modules Class Diagram
K.3  i3PersistenceApi: Module Metrics and Package Diagram

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Packages</td>
<td>5</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>8</td>
</tr>
<tr>
<td>Number of Interfaces</td>
<td>11</td>
</tr>
<tr>
<td>Total Lines of Code</td>
<td>343</td>
</tr>
</tbody>
</table>

Table K.3: i3PersistenceApi Module Metrics

Figure K.5: i3PersistenceApi Module Package Diagram
K.3.1 i3PersistenceApi: Query Module Class Diagram

Figure K.6: i3PersistenceApi Query Module Class Diagram
K.3.2 i3PersistenceApi: Modules Class Diagram

Figure K.7: i3PersistenceApi Modules Class Diagram
K.4 i3InjectionImpl: Module Statistics and Package Diagram

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Packages</td>
<td>11</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>73</td>
</tr>
<tr>
<td>Number of Interfaces</td>
<td>4</td>
</tr>
<tr>
<td>Total Lines of Code</td>
<td>3981</td>
</tr>
</tbody>
</table>

Table K.4: i3InjectionImpl Module Metrics

Figure K.8: i3InjectionImpl Module Package Diagram
K.5  i3PersistenceImpl: Module Statistics and Package Diagram

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Packages</td>
<td>6</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>61</td>
</tr>
<tr>
<td>Number of Interfaces</td>
<td>14</td>
</tr>
<tr>
<td>Total Lines of Code</td>
<td>4599</td>
</tr>
</tbody>
</table>

Table K.5: i3PersistenceImpl Module Metrics

Figure K.9: i3PersistenceImpl Module Package Diagram