# *Konzeption und Realisierung einer Datenbankanbindung für den .getmore Testfallgenerator*

Studienarbeit

***Scott Allen Hady***

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg

# Konzeption und Realisierung einer Datenbankanbindung für den .getmore Testfallgenerator

Studienarbeit im Fach Informatik

vorgelegt von

**Scott Allen Hady**

geb. 28.10.1978 in Viroqua, Wisconsin - U.S.A.

angefertigt am

**Department Informatik**
**Lehrstuhl für Informatik 6 (Datenmanagement)**
**Friedrich-Alexander-Universität Erlangen-Nürnberg**

Betreuer: Univ.-Prof. Dr.-Ing. habil. Richard Lenz
Dipl.-Inf. Christoph P. Neumann

Beginn der Arbeit: 01.07.2010
Abgabe der Arbeit: 10.02.2011

# Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Studienarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 10.02.2011

(Scott Allen Hady)

# Kurzfassung

## Konzeption und Realisierung einer Datenbankanbindung für den .getmore Testfallgenerator

.getmore ist ein automatischer Testfallgenerator, der Testfälle aus modellierten Softwaresystemen erzeugt. Er erzeugt eine Reihe von Testfällen, filtriert sie um Redundanz zu verringern und exportiert sie schließlich in eine Testumgebung, wo sie ausgeführt werden. Der Generator kann auf jeder Testebene, von Modul- bis zu Akzeptanztests, durch einen Programmierer oder ein Testteam eingesetzt werden.

.getmore speichert seine Daten als XML Datein im Dateiensystem, woraus sich eine Reihe von Schwachstellen ergeben: Es bietet keine Datensicherheit, kein explizites Backup, garantiert keine Integrität, Konsistenz oder Beständigkeit, bietet keine transaktionale Unterstützung und nur lokalen Ein-Benutzer-Zugriff. Die Verwendung einer Datenbank kann viele dieser Schwierigkeiten beheben. Dieses Projekt benutzt die eXist-db, eine native open source XML Datenbank, um die internen Datenstrukturen zu speichern. Die eXist-db bietet XML Datenpersistenz, automatische Wiederherstellung nach Abstürzen, zahlreiche Zugangsmethoden und Unterstützung für XPath, XQuery und XSLT.

Dieses Projekt implementiert ein dynamisches Persistenz-Framework, das fähig ist, die Daten von .getmore für mehrere lokale und Netzwerk-Benutzer durch vier unterschiedliche Methoden zu erhalten: Dateinsystem, eingebettete Datenbank, lokale Datenbank und Server. Ein intelligenter Ansatz für den Transport von Daten zwischen dem Dateiensystem und der Datenbank und ein System zur Versionskontrolle innerhalb der Datenbank sind vorgestellt und umgesetzt. XQuery Funktionalität ist mit Hilfe der Template Method Entwurfsmuster unterstützt. Zusätzlich ist die Performanz der eingebetteten Datenbank und der Dateinsystem-Methoden evaluiert, um sequentielle und nicht-sequentiellen Zugriffsmethoden zu vergleichen.

# Abstract

## Conception and Implementation of Database Persistence for the .getmore Test Case Generator

The .getmore system is a automatic test generation tool for supporting the testing of a modeled software system. Given a UML model of the software system, it creates a set of test cases, filters the test cases to reduce redundancy and then exports the test cases to a given testing environment for execution. It can be applied at any level of testing, from unit to acceptance testing, by an individual programmer or team of testers.

The system currently persists its internal data structures as XML files in the file system, which has a series of weaknesses to include: no data security, no explicit backup and restoration capabilities, does not guarantee the data's integrity, consistency or durability, no transactional support and only local single-user sequential access. Use of a database alleviates many of these difficulties. This project uses the eXist-db, an open source Native XML Database, in order to persist the system's internal data structures. The eXist-db provides XML data persistence, automatic recovery from crashes, numerous access methods and support for XPath, XQuery and XSLT.

This project implements a dynamically controlled persistence framework capable of persisting the .getmore's internal data structures for multiple local and remote users through four differing methods: file system, embedded database, local database server and remote database server. An intelligent approach to transporting data between the file system and the database and a version control system within the database are designed and implemented. XQuery functionality is supported using the Template Method Design Pattern. Finally, performance evaluations of the file system and embedded database methods provide a comparison of the sequential and non-sequential access methods.

# Contents

**List of Figures** **v**

**List of Tables** **vii**

**List of Abbreviations** **ix**

**1 Introduction** **1**

1.1 The .getmore System . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Project's Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2.1 File System Persistence Challenges . . . . . . . . . . . . . . . . . 3

1.3 Project's Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Method** **5**

**3 Project Requirements** **7**

3.1 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

3.1.1 R1-4: Persist Internal Data Structures in an Embedded Database 7

3.1.2 R5: Provide Local and Remote Access for Multiple Users . . . . . 8

3.1.3 R6: Provided Filtered Access to Persisted Data . . . . . . . . . . 9

3.1.4 R7-16: Extended Requirements . . . . . . . . . . . . . . . . . . . 9

3.2 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

**4 Applicable Technologies** **13**

4.1 XML and Related Technologies . . . . . . . . . . . . . . . . . . . . . 13

4.1.1 XML Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

4.1.2 XPath and XQuery . . . . . . . . . . . . . . . . . . . . . . . . . . 15

4.2 XML Storage in Databases . . . . . . . . . . . . . . . . . . . . . . . . 17

4.2.1 Storage in a NXD . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.2.2 Storage in a RDBMS . . . . . . . . . . . . . . . . . . . . . . . . 18
4.2.3 Storage Overview . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3 Database Deployment Techniques . . . . . . . . . . . . . . . . . . . . . 22
4.3.1 Embedded Database . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3.2 Database Server . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.4 Database Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4.1 BaseX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4.2 eXist-db . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4.3 Sedna . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.4.4 MySQL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.4.5 Decision Support Matrix . . . . . . . . . . . . . . . . . . . . . . 25
4.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**5 Legacy System Analysis 29**

5.1 .getmore Primary Subsystems . . . . . . . . . . . . . . . . . . . . . . 29
5.1.1 The Core Subsystem . . . . . . . . . . . . . . . . . . . . . . . . 29
5.1.2 The GUI Subsystem . . . . . . . . . . . . . . . . . . . . . . . . 31
5.1.3 The Persistence Subsystem . . . . . . . . . . . . . . . . . . . . . 32
5.2 Persistence of .getmore's Internal Data Structures . . . . . . . . . . . 32
5.2.1 Storing Internal Data Structures . . . . . . . . . . . . . . . . . . 35
5.2.2 Retrieving Internal Data Structures . . . . . . . . . . . . . . . . 37
5.3 Loading, Converting and Dumping Persisted Data . . . . . . . . . . . . 39
5.3.1 Dumping Persisted Data to the File System . . . . . . . . . . . . 39
5.3.2 Loading Data from the File System into Persistence . . . . . . . . 39
5.3.3 Converting Outdated Models . . . . . . . . . . . . . . . . . . . . 40
5.3.4 Loading Models into .getmore . . . . . . . . . . . . . . . . . . . 40
5.4 Dependency on java.io.File . . . . . . . . . . . . . . . . . . . . . . . . 41
5.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**6 Development – Iteration One 43**

6.1 An Agile Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.2 Scope of Iteration One . . . . . . . . . . . . . . . . . . . . . . . . . . 44

6.3 Design – Overview . . . . . 44
6.3.1 Overarching Concerns . . . . . 44
6.3.2 Starting Basis . . . . . 45
6.4 Design – java.io.File Replacement . . . . . 46
6.4.1 Storage Component CRC Card . . . . . 47
6.4.2 Storage Component Class Diagram . . . . . 49
6.5 Design – Validation Facilities . . . . . 50
6.6 Design – Model Conversion . . . . . 51
6.7 Design – Database's Internal Structure . . . . . 52
6.8 Design Result Overview – Persistence Subsystem . . . . . 53
6.9 Implementation – Proceeding and Selected Details . . . . . 53
6.10 Objectives Accomplished in Iteration One . . . . . 57
6.11 Summary . . . . . 57

**7 Development – Iteration Two 59**

7.1 Scope of Iteration Two . . . . . 59
7.2 Design – Overview . . . . . 60
7.3 Design – Loading, Converting and Dumping Models . . . . . 61
7.3.1 Flexible Data Transport . . . . . 61
7.3.2 Model Dumping . . . . . 62
7.3.3 Model Converting and Loading . . . . . 63
7.4 Design – Local and Remote Access . . . . . 65
7.4.1 Dynamic Persistence Framework . . . . . 66
7.4.2 Persistence Properties . . . . . 67
7.4.3 Fault-Tolerant Persistence . . . . . 68
7.4.4 Persistence Directory . . . . . 70
7.5 Design – Separation of User and System Concerns . . . . . 71
7.6 Design – Support for Future .getmore Version . . . . . 72
7.6.1 UID Support . . . . . 72
7.6.2 Extended Data Structure Support . . . . . 73
7.7 Design – Model Version Control . . . . . 74
7.7.1 Versioning Design Decisions . . . . . 74
7.7.2 Versioning Design . . . . . 75

7.7.3 Versioning Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
7.7.4 Versioning Actions . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
7.8 Design – XQuery Template . . . . . . . . . . . . . . . . . . . . . . . . . . 80
7.9 Design – Administrative Client . . . . . . . . . . . . . . . . . . . . . . . . 81
7.10 Design Result Overview – Persistence Subsystem . . . . . . . . . . . . . . 81
7.11 Implementation – Proceeding and Selected Details . . . . . . . . . . . . . . 82
7.12 Objectives Accomplished in Iteration Two . . . . . . . . . . . . . . . . . . 85
7.13 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

**8 Project Evaluation** **89**
8.1 Software Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
8.2 Performance Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
8.3 Developmental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 91
8.4 Discussion and Future Work . . . . . . . . . . . . . . . . . . . . . . . . . 92
8.4.1 Filtering Mechanism . . . . . . . . . . . . . . . . . . . . . . . . . . 92
8.4.2 Transactional Support . . . . . . . . . . . . . . . . . . . . . . . . . 93
8.4.3 Version Control Subsystem . . . . . . . . . . . . . . . . . . . . . . . 93
8.4.4 Other Areas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
8.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

**9 Conclusion** **95**

**Appendices** **97**

**A Project Requirements Purpose Analysis** **97**
A.1 R1-4: Persist Internal Data Structures in an Embedded Database . . . . 97
A.2 R5: Provide Local and Remote Access for Multiple Users . . . . . . . . . 98
A.3 R6: Provide Filtered Access to Persisted Data . . . . . . . . . . . . . . . 98
A.4 R7-16: Extended Requirements . . . . . . . . . . . . . . . . . . . . . . . 98

**Bibliography** **I**

# List of Figures

1.1 The .getmore System . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 The File System Persistence Hierarchy . . . . . . . . . . . . . . . . . . . . 3

4.1 XML-to-Relational Mapping for Storage and Retrieval . . . . . . . . . . 19
4.2 XML Storage Overview . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3 Embedded Database and Database Server . . . . . . . . . . . . . . . . . 23

5.1 .getmore Primary Subsytems . . . . . . . . . . . . . . . . . . . . . . . 30
5.2 .getmore GUI Screenshot . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3 Persistence Subsystem Data Access Objects . . . . . . . . . . . . . . . . 33
5.4 Persistence Subsystem Class Diagram . . . . . . . . . . . . . . . . . . . 34
5.5 Storing Internal Data Structures Overview . . . . . . . . . . . . . . . . 35
5.6 Model Storage Sequence Diagram . . . . . . . . . . . . . . . . . . . . . 36
5.7 Retrieving Internal Data Structures Activity Diagram . . . . . . . . . . . 37
5.8 Model Retrieval Sequence Diagram . . . . . . . . . . . . . . . . . . . . 38
5.9 Model Loading and Conversion Sequence Diagram . . . . . . . . . . . . . 41

6.1 Storage Component CRC Card . . . . . . . . . . . . . . . . . . . . . . 48
6.2 StorageComponent Class Diagram . . . . . . . . . . . . . . . . . . . . . 50
6.3 XSValidator Class Diagram . . . . . . . . . . . . . . . . . . . . . . . . 51
6.4 eXist-db Collection Structure . . . . . . . . . . . . . . . . . . . . . . . 52
6.5 Persistence Subsystem's Class Diagram – Iteration One . . . . . . . . . . 53
6.6 The Peristence Subdirectory Structure . . . . . . . . . . . . . . . . . . . 54
6.7 eXist-db Access Activity Diagram . . . . . . . . . . . . . . . . . . . . . 55
6.8 Resulting .getmore System – Iteration One . . . . . . . . . . . . . . . . 57

7.1 Model Loading and Dumping Class Diagram . . . . . . . . . . . . . . . . 62

7.2 Model Dumping Class Diagram . . . . . . . . . . . . . . . . . . . . . . . 63
7.3 Model Conversion and Loading Class Diagram . . . . . . . . . . . . . . . . 65
7.4 Database Persistence Methods Overview . . . . . . . . . . . . . . . . . . . 66
7.5 PersistenceProperties Class Diagram . . . . . . . . . . . . . . . . . . . . . 68
7.6 PesistenceInfo Creational Activity Diagram . . . . . . . . . . . . . . . . . 69
7.7 The Peristence Subdirectory Structure . . . . . . . . . . . . . . . . . . . . 71
7.8 Refactored Interfactes to Support UID Identification . . . . . . . . . . . 73
7.9 Extended ITestCaseTreeStateInfo Interface . . . . . . . . . . . . . . . . . 73
7.10 Versioning State Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . 76
7.11 Versioning System and Trigger CRC Cards . . . . . . . . . . . . . . . . . 77
7.12 Versioning Subsystem Layout . . . . . . . . . . . . . . . . . . . . . . . . 79
7.13 Model Versioning Activity Diagram . . . . . . . . . . . . . . . . . . . . . 79
7.14 eXist-db Java Administrative Client . . . . . . . . . . . . . . . . . . . . . 81
7.15 Persistence Subsystem's Class Diagram – Iteration Two . . . . . . . . . . 82
7.16 Resulting .getmore System – Iteration Two . . . . . . . . . . . . . . . . . 86

# List of Tables

4.1 Decision Support Matrix (Lowest Result is Best) . . . . . . . . . . . . . . . 26

5.1 Persistence Subsystem and java.io.File Coupling . . . . . . . . . . . . . . . . 42

7.1 Data Transfer Target Combinations . . . . . . . . . . . . . . . . . . . . . . . . 61

8.1 Performance Comparision Results . . . . . . . . . . . . . . . . . . . . . . . . 91

# List of Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **BLOB** | Binary Large OBject |
| **BSD** | Berkley Software Distribution |
| **CLOB** | Character Large OBject |
| **CRC** | Class-Responsibilities-Collaborators |
| **CVS** | Concurrent Versions System |
| **DAO** | Data Access Object |
| **DOM** | Document Object Model |
| **DTD** | Document Type Definition |
| **GNU GPL** | GNU's Not Unix General Public License |
| **GNU LGPL** | GNU's Not Unix Lesser General Public License |
| **GUI** | Graphical User Interface |
| **HTTP** | Hypertext Transfer Protocol |
| **JAR** | Java Archive |
| **JAXP** | Java API for XML Processing |
| **JAXV** | Java API for XML Validation |
| **JDBC** | Java Database Connectivity |

| | |
|---|---|
| **JVM** | Java Virtual Machine |
| **NXD** | Native XML Database |
| **RDBMS** | Relational Database Management System |
| **REST/JAX-RX** | Unified Representational State Transfer Access to XML Resources |
| **REST** | Representational State Transfer |
| **SAX** | Simple API for XML |
| **SGML** | Standard Generalized Markup Language |
| **SOAP** | Simple Object Access Protocol |
| **SQL** | Structured Query Language |
| **StAX** | Streaming API for XML |
| **SUT** | System Under Test |
| **SVN** | Subversion |
| **TDD** | Test Driven Development |
| **UID** | Unique Identifier |
| **UML** | Unified Modelling Language |
| **URI** | Uniform Resource Identifier |
| **W3C** | World Wide Web Consortium |
| **WebDAV** | Web-based Distributed Authoring and Versioning |
| **XML** | Extended Markup Language |
| **XML:DB API** | XML Database API |
| **XML-RPC** | XML-Remote-Procedure-Call |
| **XQJ** | XQuery for Java |

| | |
|---|---|
| **XP** | eXtreme Programming |
| **XPath** | XML Path Language |
| **XQuery** | XML Query Language |
| **XSD** | XML Schema |
| **XSLT** | Extensible Stylesheet Language Transformations |

# 1 Introduction

Software systems have infiltrated every aspect of today's life fulfilling everything from simple entertainment to safety-critical tasks such as air traffic control. The software development industry is correspondingly developing larger and more complex systems in order to meet the ever-increasing demands of businesses as they fight to provide the "latest-and-greatest" application to satisfy the the consumer. The increasing complexity of software being developed requires also a corresponding advancement of the software's testing methods in order to ensure that it fulfills the customers' desires and expectations. sepp.med GmbH's .getmore system[1] strives to provide this advancement through an integrated model-centric test case development system.

## 1.1 The .getmore System

The .getmore (GEnerating Tests for MOdels to Reduce Effort) system generates test cases from modeled software systems in order to simplify and automate the systematic generation of scenarios for integration, system and acceptance tests. First the desired system to test is modeled in a modeling environment such as Enterprise Architect (EA) or Artisan Studio as a Unified Modelling Language (UML) diagram. The created Model is then imported into the .getmore system. A testing strategy such as full path or named path coverage is then applied to the internal representation of the system and generates a complete listing of all possible test cases that fulfill the designated strategy, referred to as a Test Case Tree (TCT). The complete listing may have redundancies. These redundancies may then be filtered out, either manually or automatically, to produce a recommended Test Case Tree State (State), which is a minimal set of test cases

1 .getmore Homepage: `www.seppmed.de/produkte/getmore.html`

that accomplish the tester's goals. These test cases may then be exported to a test management tool for their execution. Figure: 1.1 depicts the overall .getmore system.

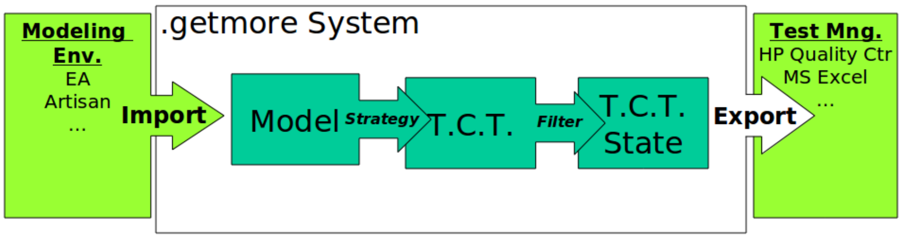


**Figure 1.1:** The .getmore System

The .getmore system automatically generates and manages coverage test cases for large or small software systems through various path and node strategies. This frees software testers from the mundane task of developing coverage test cases and allows them to focus the special application fringe test cases, where difficult to diagnose "bugs" may reside. The combination of automatically generated coverage test cases and "expert" generated specialized test cases provides significantly better coverage than either approach could alone.

## 1.2 Project's Motivation

The .getmore system currently persists its data in files stored, transparently to the user, in the file system. The persistent data is broken into a directory hierarchy that matches internal system data structure: Model, Test Case Tree and Test Case Tree State. The majority of the data is persisted in Extended Markup Language (XML) files, which provides the flexibility to describe the different Models while still remaining human-readable. However some additional data pertaining to the display of the data structures in the Graphical User Interface is stored in separate non-XML files.

The Model is stored in a self-named directory containing the data describing the System Under Test's architecture as well as a number of directories, each containing Test Case Trees generated for the Model. Each Test Case Tree is stored in a self-named directory (contained within respective Model directory) and contains the data describing the generated tree of test cases as well as a number of directories each containing the saved states of the Test Case Tree. Each Test Case Tree State is stored in a self-named directory (contained within respective Test Case Tree directory) and contains the data describing the stored state of its respective Test Case Tree. Figure: 1.2 depicts the persistence hierarchy used to store the internal data structures.

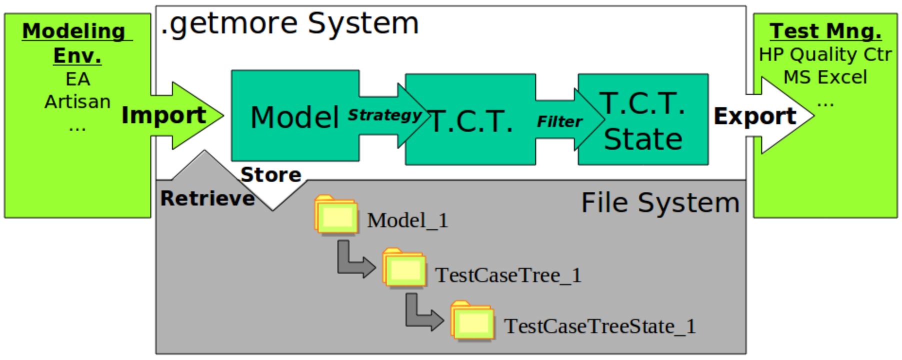


**Figure 1.2:** The File System Persistence Hierarchy

### 1.2.1 File System Persistence Challenges

The current file system persistence subsystem is a simple straightforward method with a number of limitations. Without controlling the authentication, authorization and access of the user manipulating the data, there is no way to prevent the unwanted tampering of the persisted data. A user may simply or unwittingly delete, overwrite or otherwise corrupt the data. It also provides no guarantee that the data remains permanent, consistent and uncorrupted. Lacking explicit backup and restoration capability, means there is no defense against the corruption or loss of data. The file system cannot guarantee that the data is correctly and completely stored, because it has no transactional support. The sequential only access to data limits the file system's capability to directly query or update individual elements without sequential processing the entire document in which the element is held. This can lead to significant waste of resources while searching for specific elements against specified criteria.

The .getmore system can only support access to the persisted data locally for a single user. This can be extremely limited for the target audience of a team of quality assurance specialists working in unison to test a software system. If employed in this desired scenario it creates a bottleneck in the testing process, reducing its usefulness. The data should be accessible from an arbitrary set of workstations locally or remotely connected.

Only a single version of a Model may be maintained within the system. Any updates to the system overwrite the previous version. This creates a dangerous “no way back” environment that does not enable efficient development and refinement of the system model and test cases.

All automatic state generation is done through complex filtering algorithms with Models and Test Case Trees completely loaded into the system’s memory. The sequential processing created by file processing makes the overall filtering inefficient. It would be much more efficient to operate only on data that meets specified criteria.

## 1.3 Project’s Goals

The primary goal of the project is to replace the file system persistence subsystem with a database implementation. By using a persistence subsystem based on a database, the system will have greater control and security over its data, providing more stable performance by preventing inadvertent tampering with the stored data. A database also provides measures to ensure the data is manipulated atomically as well as maintains its consistency, integrity and durability. Furthermore, a database provides non-sequential and filtering access to the data through its querying methods.

The secondary goal of this project is to provide local and remote access for multiple users to a common set of data. This flexibility is critical for its effective use by its target audience, the software quality assurance team. It will allow multiple team members to manipulate a common set of data from a variety of platforms and locations. By working simultaneously on a communal set of data they will maintain a common perception of the System Under Test and the progress of their efforts.

The tertiary goal of this project is to provide a filtering mechanism to reduce the amount of data that the system must process in order to automatically filter the generated Test Case Tree. This project is not expected to accomplish this goal but it should provide the groundwork for its later implementation.

The desired end state of this project is that the .getmore system’s data is persisted in a database that provides local and remote access to a common set of data for multiple users simultaneously.

# 2 Method

First an initial analysis of the product's requirements was conducted. An initial set of requirements was established, discussed, refined and prioritized based on the expected needs of the system. These requirements were continually refined and reprioritized throughout the development of this product to represent the most current perspective of the client. For a more thorough discussion of the product's requirements and their management see Chapter 3: *Project Requirements* on page 7.

Once the initial set of requirements were established, the possible technologies were then analyzed for their suitability in fulfilling the product's requirements. Various XML technologies such as: XML Schema, Simple API for XML parsing, Document Object Model parsing, XML Path Language and XML Query Language were investigated. Database technologies for the storage of XML and employment were also researched. Based on these initial requirements it was important to determine a suitable database for the implementation. The database should have an non-proprietary license, be stable, be well documented, be capable of being automatically integrated into the system's installation, require little maintenance as the system's structure (schemas) remained fluid and support XML manipulation and querying. Four databases were analyzed and presented for acceptance. For a more thorough discussion of the applicable technologies see Chapter 4: *Applicable Technologies* on page 13.

Next began a major effort of analyzing the legacy .getmore system, which would be upgraded. First the source code and supporting libraries were accessed from the client's Subversion server. Second the user's manuals were read and a basic tutorial of the system were completed. Next an analysis of the overall system, its primary classes and the typical flow of control were conducted. Finally an in depth analysis of the persistence subsystem was conducted. Special notice was taken of the contracts made by the persistence layer's interface, as they were to remain intact to allow the simple exchange between persistence subsystems. The relationship between the persistence subsystem's Data Access Objects and the system's primary objects was also discovered. For more details on the analysis of the legacy system, see Chapter 5: *Legacy System Analysis* on page 29.

This project was implemented in two one-month iterations, not considering the initial research and documentation efforts; though these efforts did continue throughout the development, they were not the primary focus. For each of the iterations, a set of requirements was agreed upon and solidified prior to the start of each iteration. For more details on the general development effort, see Section 6.1: *An Agile Approach* on page 43.

The priority effort for the first iteration was the accomplishment of the project's first and primary goal: Replace the file system dependant persistence subsystem with a database implementation. Based on the analysis of the legacy system, a design was developed to achieve this goal and then the design was implemented. For more details on the first iteration, see Chapter 6: *Development – Iteration One* on page 43.

The priority effort of the second iteration was the project's secondary goal: Provide multiple user access to communally persisted data. Beyond that the Model's loading, converting and dumping subcomponent was updated; the persistence subsystems were updated to support the expected needs for a future .getmore version; a Model versioning system was created and the groundwork for a filtering mechanism was created and tested for performance gain. Once again a design to achieve these goals was created and then implemented. For more details on the second iteration, see Chapter 7: *Development – Iteration Two* on page 59.

Upon completion of the second iteration, the project was evaluated and recommendations were made for its continued development. The product was evaluated against basic software metrics and performance. The overall developmental process was evaluated to provide a perspective for future improvements. Each feature was also reviewed and a recommendation for continued effort was provided if needed to ensure that the receiving developer would have an appropriate "place-to-start". For more details on the provided recommendations, see Chapter 8: *Project Evaluation* on page 89.

# 3 Project Requirements

In this chapter requirement management will be described following by a definition of the requirements accumulated during the execution of this project. For an analysis of the requirements' purposes, see Appendix A: *Product Requirement Purpose Analysis*.

An incremental and iterative agile method of development, employing aspects of Scrum, was employed during this project and thus the requirements were managed in two lists. Desired requirements were initially added to a dynamic "Backlogged Feature List" and requirements to be implemented in the current iteration were agreed upon, refined and added to a stable "Active Feature List" prior to the start of the iteration. These lists are similar to Scrum's "Product Backlog" and "Sprint Backlog" respectively. This provided the necessary flexibility for this project's prototypical development and eased the transition between the client's responsible parties.

## 3.1 Requirements

In this section each of the requirements will be investigated and detailed. These requirements are the basis against which the success of this project will be judged. See Appendix A: *Product Requirement Purpose Analysis* for more details describing the purpose of each requirement.

### 3.1.1 R1-4: Persist Internal Data Structures in an Embedded Database

The persistence subsystem must persist the .getmore system's internal data structures, defined as each Model with its respective Test Case Trees and their respective Test Case Tree States, and their supporting data structures, defined as the respective metadata, version, lock and expansion data, within an embedded database and returned them to the system on demand. The embedded database persistence subsystem must use the same interface as the current file system persistence subsystem and fulfill the same interfacial contracts. The embedded database persistence subsystem must also maintain the same persistence transparency, meaning that the user should not be required to provide additional configuration information in order to use the persistence subsystem.

**R1: Access the embedded database.** The persistence subsystem must provide transparent access to the embedded database. The appropriate addressing, user name and password must be maintained within the subsystem.

**R2: Store the internal data structures in embedded database.** The subsystem must persist the designated system data within the embedded database using the same persistence interface as the file system subsystem.

**R3: Retrieve the internal data structures from embedded database.** The subsystem must retrieve and validate the data on demand using the same persistence interface as the file system subsystem.

**R4: Integrate the embedded database persistence subsystem into the .getmore system.** The .getmore system should remain ignorant to the fact that an alternative subsystem is being used to persist the data and all of the functionality provided by the file system persistence subsystem should also be provide by the new subsystem. Additionally, the subsystem's documentation and exception handling should be implemented in accordance with the directives of the .getmore system.

### 3.1.2 R5: Provide Local and Remote Access for Multiple Users

The persistence subsystem must provide both local and remote access for multiple users to a common set of persisted data through the use of a database in server mode. A single database, configured as the server, must persist the data accessed and manipulated by any number of client systems. The clients must access the database server through a normal Internet connection. The address to the server, user name and password must be provided to the clients and maintained in the persistence subsystem. Other than the additional configuration parameters (server address, user name and password) the

.getmore system and user must remain ignorant of any differences between the persistence subsystem, which must all employ and fulfill the same interfacial contracts.

### 3.1.3 R6: Provided Filtered Access to Persisted Data

The persistence subsystem must filter data retrieved from the persistence subsystem. The filtering must be done with respect to the criteria set forth by the system's filtering subsystem, which generates Test Case Tree States from a given Test Case Tree. A limited set of data must be returned through the use of the database's querying mechanisms. The persistence interface may be extended in order to accomplish this requirement. It is not expected that this project fulfill this requirement; only that it provides the groundwork for its later implementation.

### 3.1.4 R7-16: Extended Requirements

The following requirements form an extended set of features that bring additional value directly to the project, but were not identified as part of the critical base set during the initial project analysis or were discovered during the execution of the project.

**R7: Provide Model Versioning Control.** The persistence subsystem must provide the ability to store revisions or changes of the data structures and on demand return the data structures to a previous revision.

**R8: Integrate the Database into .getmore's Installation Process.** The persistence subsystem's configuration should be integrated into the .getmore system's automatic installation process. There should be no additional configuration effort required for the use of the embedded database persistence subsystem and there should be a very minimum of additional effort required to provide remote access to the persisted data.

**R9: Convert Models Created by other .getmore Versions.** The persistence subsystem must automatically determine the version of each Model, in persistence during startup or while loading new Models, and attempt to rejuvenate the persisted data to match the current .getmore system's version. Any Model, which cannot be converted, must be stored in an archive and removed from the .getmore system to allow for a more focused conversion effort and prevent mishandling of data structures. The end-user must be able to upgrade to a new system version and still continue working with the Models generated in older versions.

**R10: Dump Persisted Models to File System.** The persistence subsystem must dump designated Models from the database into a designated file system directory in the same format as expected by the original file system persistence subsystem. The persistence subsystem must be able to dump either all Models or a just designated set of Models.

**R11: Load Models from File System into the Database.** The persistence subsystem must load designated Models from a designated file system directory into the database. The Models must be presented in the same format used by the original file system persistence subsystem. The user is responsible for ensuring the data to be loaded is correctly formatted and organized. The subsystem must be able to load either all Models or just a designated set of Models.

**R12: Provide Transactional Support.** The persistence subsystem must provide transactional support for database persistence operations. The persistence subsystem must ensure the atomicity of the transactions allowing the .getmore system to begin a transaction and then commit, rollback or abort that transaction as desired. It is not expected that this project fulfill this requirement; only that it provides the groundwork for its later implementation.

**R13: Provide Backup and Restoration of Persisted Data.** The persistence subsystem must on demand backup the persisted data and on demand restore the data from a backed up data store. The data must be backed up into a local file system directory and should provide the option for the user to specify where the data is backed up. Any number of backups must be able to be stored and the user must be able to choose which backup to restore. It is not expected that this project fulfill this requirement; only that it provides the groundwork for its later implementation.

**R14: Provide a Database Administrative Client.** The administrative client must allow an administrator to log in to the database and perform typical maintenance operations, such as managing user accounts, directly manipulating the data stored in the database, backing up and restoring data. It is not expected that this project fulfill this requirement; only that it provides the groundwork for its later implementation.

**R15: Support UIDs for Element Identification.** The persistence subsystems must use Unique Identifier (UID) strings generated by the .getmore system to identify the persisted data structures instead of the current name based identification. The system must request elements for retrieval through their UID.

**R16: Support an Extended Internal Data Structure.** The persistence subsystems must be adjusted in order to facilitate the storage of a new internal data structure. Test Case Trees must be able to contain either other Test Case Trees or Test Case Tree States. Test Case Tree States must be able to contain either Test Case Trees or other Test Case Tree States.

**RX: Provide Future Developmental Recommendations.** Any requirements that are not implemented during the time frame of this project should be assessed for their estimated impacts. Recommendations for possible implementation during future development must also be included in the technical report.

## 3.2 Summary

In this chapter the process of requirements discovery and management was described. Then each discovered requirement was defined. A total of 17 requirements were identified during this project. Of which 13 were to be completed and four were designated as future developmental efforts.

# 4 Applicable Technologies

In this chapter a brief overview of several technologies that may be applied to alleviate the system's challenges will be given. First XML and several related technologies will be investigated. Next the topic of XML storage in databases and possible database deployment techniques will be covered. Finally a synopsis of analyzed databases and their comparison will be presented. See designated references for further information on these topics.

## 4.1 XML and Related Technologies

The World Wide Web Consortium (W3C)[1], an organization focused on the promotion of interoperability standards of the web, introduced the Extended Markup Language (XML) 1.0 as a Recommendation in 1998. The primary object of XML was to provide a simple, platform-independent standard for exchanging semi-structured text based data between various participating applications. XML 1.0 provides a set of rules for creating and processing a well-formed XML document. XML syntax borrowed heavily from its predecessor Standard Generalized Markup Language (SGML) [ISO86]. XML documents are text documents consisting of semi-structured data organized by markup tags. The data is organized into the content, and the metadata that provides information about the content. The element is the basic unit of XML and consists of a start tag, its content and an end tag. The start tag contains the element's name and attributes, which are name value pairs of metadata. The end tag is identified through the element's name. Elements may contain other elements as well as textural content [BPSM$^+$08][VV06][Sar06].

The key syntactical rules for a "well-formed" XML document are:

- There is a single root element, which contains all other elements.

1 W3C Homepage: `www.w3.org`

- The start, end and empty element tags, which delimit the elements, may be nested but must maintain proper scope, meaning that none are missing or overlapping.
- There is only properly encoded legal Unicode characters and the special syntax characters ('<', '>' and &) only appear as syntactically expected.

An XML document's grammar is a description of the elements, attributes and the relationship between them. The grammar can be described through a Document Type Definition (DTD), an XML Schema (XSD) or a number of other languages. DTD was the initial method of grammar definition associated with XML 1.0 and is thus widely used. It is very compact but lacks support for some of the new features added to XML, such as namespaces and data types. XSD was developed in response to the limitations of the DTD. It follows XML syntax and provides support for both namespaces and data types. However, it is much more verbose than a DTD. An XML document that is well formed and follows the rules defined by a grammar is considered "valid" [BBC+98][FW04][VV06].

### 4.1.1 XML Parsing

Parsing is the process of extracting the information contained in XML document. There are two primary categories of parsing Application Programming Interfaces (APIs): stream-orienting and tree-traversal APIs. Stream-oriented parsing APIs use callback functions to respond to events as the parser encounters them. They require less memory, parsing time and developmental effort; but are limited to the sequential processing of the document. Tree-traversal parsing APIs first build a representation of the document in memory and then retrieve information by traversing the represented document. They require more memory, parsing time and developmental effort; but do allow for the non-sequential access and processing of the document [VV06].

Stream-oriented parsing can be accomplished through using the Simple API for XML (SAX) processing or Streaming API for XML (StAX) processing. SAX parsing is considered the push method of stream-oriented parsing, because the parser "pushes" the events, which are generated as the parser serially processes the contents, though callback functions of a designated content handler object. The handler then executes the appropriate action necessary to respond to the event's occurrence. Since the content is not maintained in memory, this processing model uses little memory and is fast. However, it can only read the data serial and is unable to modify the data contained in the XML documents. StAX parsing applies a very similar approach, except that the application controls the production of event generation and has the ability to modify content within the processed document. The StAX parser provides either a cursor or iterator to the application, which is then used to control the flow of events. The cursor and iterator approach are generally equivalent and provide the same basic capabilities. StAX is best suited for processing content that is being streamed over a network connection [Meg05][CC09].

Document Object Model (DOM) parsing is a tree-traversal API, which first builds an in-memory representation of the entire document and then provides traversal and access methods for determining and manipulating the content of the document. DOM processing is slower and requires more memory than the stream-oriented processing models, but does allow for non-sequential and repeated access and manipulation of the contents [Le 04].

### 4.1.2 XPath and XQuery

XML Path Language (XPath) is a technology used to address distinct nodes within an XML document. XPath 2.0 was introduced as a W3C Recommendation in on January 23, 2007. The XPath data model views the content of a document as a tree of nodes. Since there can only be a single root node and XML is inherently ordered, each node within this tree can be uniquely addressed from the root even if there are multiple nodes at the same level with the same name. An address, known as an XPath expression, describes a sequence of steps that must be taken in order to arrive at the desired node. Each step moves along one of the 13 axes and may be restricted by a node test and an optional predicate. Each step in the expression is delineated through a slash (/) and the first slash represents the root node [CD99].

Listing: 4.1 is an XPath expression, which moves two steps along the child axis to a node named author and has textural content that is equal to "`Ajay Vohra`".

```
/child::book/child::author[text() = "Ajay␣Vohra"]
```

**Listing 4.1:** XPath Expression Example

XML Query Language (XQuery) is a functional language that provides a means to query the content of an XML document, similar to the querying functionality that the Structured Query Language (SQL) provides for data stored in a Relational Database Management System (RDBMS). XQuery 1.0 is based on XPath 2.0 and became a W3C Recommendation on January 23, 2007. While they are similar there are a couple of fundamental differences between XQuery and SQL. First, SQL operates on an unordered set of data whereas XQuery operates on XML, which is inherently ordered. SQL is specifically designed for operating on a very strictly defined set of relational data from a single data source, but XQuery is designed to operate on loosely defined XML data from possibly multiple data sources. SQL has defined functionality for the manipulating or changing of the data on which it operates, however, XQuery provides only querying functionality [BCF$^+$07][CFL$^+$99].

XQuery's basic statement clausal structure is referred to as a For Let Where Ordered by Return (FLWOR) expression, which is the general order in which the clauses occur. The `for` clause associates one or more variables to expressions. The evaluation of the expression creates a tuple stream and these tuples are bond one at a time to the variables. This is similar to a `for` statement in a procedural language. The `let` clause is similar to the `for` clause except that it binds the entire result of an expression to a variable. The `where` clause is used to filter the tuples returned by the evaluated expressions. The `order by` clause sorts the retrieved tuples in the stream. The `return` clause creates the result of the query for a given tuple [Kat04]. Multiple XQuery's may be chained to produce complex queries, because the input for each clause is one of the XPath 2.0 data model and the output is also one of the same data model. Therefore the output of one query may be used as the input of the next. The following is a summary of an example, depicted in Listing: 4.2 on the next page, provided by the W3C and the following description parallels the provided W3C description [BCF$^+$07].

This XQuery queries simultaneously two separate XML documents. The first describes the departments in a company and the second describes the employees of that company. The `for` clause iterates over all of the departments, binding the variable `$d` to each department in turn. The `let` clause binds the variable `$e` to all of the employees that belong to the department for which the variable `$d` is currently bound. The combination of the binding of the variables `$d` and `$e` to the results of their respective expressions creates a stream of tuples, or sets of data. The `where` clause filters departments with

```
for $d in fn:doc("depts.xml")/depts/deptno
let $e := fn:doc("emps.xml")/emps/emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
    <big-dept>
        {
        $d,
        <headcount>{fn:count($e)}</headcount>,
        <avgsal>{fn:avg($e/salary)}</avgsal>
        }
    <big-dept>
```

**Listing 4.2:** XQuery Statement Example

less than 10 employees out of the tuple stream. The `order by` clause then sorts the remaining tuples in the tuple stream descending according to the average salary of the employees. The `return` clause then builds the results describing the large departments, their number of employees and the employees average salary [BCF+07].

## 4.2 XML Storage in Databases

Storage of XML in a database can be accomplished in many ways depending the database implementation being used. While there are many other options, only storage in a Native XML Database (NXD) and Relational Database Management System (RDBMS) will be discussed. These are the two databases commonly used to store XML data.

### 4.2.1 Storage in a NXD

Native XML Databases (NXDs) where introduced with the specific purpose of storing and retrieving XML documents. The XML documents, referred to as resources, are grouped into collections, similar to directories or folders in a file system. Collections may contain resources or other sub-collections. This approach has the advantage of maintaining the logical structure of the entire XML document no just the structure of the data contained within the document and alleviates the mapping challenges discussed in the next section. Additionally, the XML documents stored are not constrained by a restricting schema, thereby allowing a flexible mix of any well-formed XML documents

[VV06]. This flexibility is useful when storing multiple XML documents with varying structures.

### 4.2.2 Storage in a RDBMS

XML may be stored in RDBMSs by either mapping the XML content to a relational model and storing the mapped data in standard relational tables as tuples or by storing the document in a column of a table as an XML data type. Using a RDBMS provides the advantages accumulated other decades of experience and development. RDBMSs are very stable, scalable and have a proven track record of data management.

Many RDBMSs have been extended to handle the XML data type. These RDBMSs are commonly referred to as XML-Enabled RDBMSs. Many XML-Enabled RDBMSs also offer basic querying and updating functionality for the XML documents stored as this data type. SQL:2008 (ISO/IEC 9075-14:2008) defines the use and manipulation of XML as a data type within SQL [ISO08]. Even if the RDBMS does not support an XML data type, the document may still be stored within a Character Large OBject (CLOB) or Binary Large OBject (BLOB), obviously this is a naive approach which limits the ability to manipulate and query the stored document.

The XML and relational data models are two different data models and suffer from the typical impedance mismatch associated while trying to translate data between differing data models. In order to take advantage of the benefits of the RDBMS a “mapping” technique may be used. Mapping refers to the translation between the XML and relational data models. In order to store an XML document, the XML data must be translated into a set of tuples that may be stored in the RDBMS’s relations or tables. To retrieve the data, a query against the the XML model must be translated to an relational SQL statement which is executed and the response must be translated into the XML data model. Figure: 4.1 on the facing page depicts this mapping layer. On the left side the XML data is being stored an on the right side the data is being retrieved. Prior to storing XML data, an XML grammar may optionally be used to generate the relations in which the documents will be stored.

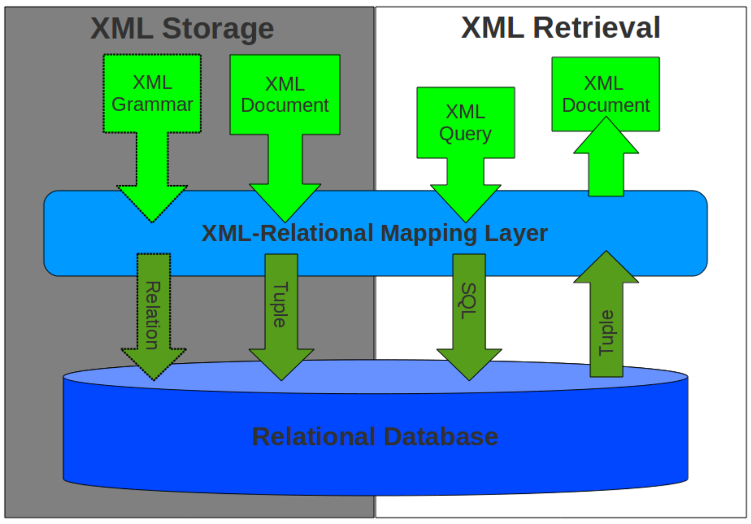


**Figure 4.1:** XML-to-Relational Mapping for Storage and Retrieval

### XR Mapping Methods

Many different methods for overcoming the XML-Relational impedance mismatch have been developed. Three common XML-to-Relational mapping categories are: Generic, Schema-Driven and User-Defined.

**Generic Mappings.** Generic mappings do not use an XML grammar to define the relations prior to storing the XML data. These approaches use the information found within the XML document in order to build a coherent relationship between the elements. These methods may further be broken into further categories: generic-tree, structure-centered and simple-path. These methods model the XML document after the DOM data model and then store this structure within the respective relations. The advantage of these methods is that they do not need an XML grammar and may be employed to store many different XML document structures. However, they may lack some of the specialized optimizations achieved through the use of the XML grammar.

*Generic-tree mapping* methods represent the XML document as a tree and then map it to a relational structure. The Edge mapping method is a common generic-tree mapping method. This method stores all edges of the tree in a relation with the following schema: `Edge(source, order, name, flag, target)`. The source is the parent node, the target is the child node, the order identifies the child's ordering among siblings, the name is the name of the edge and the flag indicates if it is an interior or leaf node. The attribute and universal mapping methods are also similar techniques.

*Structure-centered mapping* method is similar but maps the tree to the tuple `Node(type, name, content, children)`. Type refers to the type of the node (e.g. `ELEMENT`, `ATTRIBUTE`, `TEXT`,...)and the name is the name of the node. The foreign key and depth-first are two strategies that employ this method. The foreign key strategy ensures that each node has a unique identifier and maintains a foreign key reference to its parent. The depth-first strategy processes the tree in a depth-first manner and numbers the nodes as it first encounters them with a minimum and provides a maximum as it returns through the node. This allow for efficient searching based on the node ranges or intervals.

*Simple-path mapping* method uses the XPath location to uniquely identify a node and to simplify querying the data. However, since XPath mappings do not maintain the ordering among siblings the relative positional information must be maintained for each element. This method uses four relations. One for the path expressions, one for the elements, one for the attributes, and one for the text.

**Schema-Driven Mappings.** Schema-driven mappings use an XML grammar or schema to create a relational schema which is used to store valid XML documents. This is accomplished by creating one relation per element and with the element's attributes mapping to fields within the relation. Subelement references are also maintained as foreign keys within the relation. Subelements that occur once may be in-lined into the parent element. If a subelement may occur more than once, it must be mapped by a relationship table, which maintains the key-foreign key relationship. If ordering is to be maintained it may be mapped to a special column. The reconstruction of elements is accomplished by multiple table joins. Schema-driven mappings may be further categorized based on a number of different categories to include: fixed, flexible, and intermediate mappings.

*Fixed mapping* methods is a group of three algorithms that map a DTD to a relational schema. They are the Basic, Shared and Hybrid methods. They accomplish this by representing the DTD as a directed graph. Elements, attributes and operators (`?` or `*`) are represented as nodes within the graph. Then this graph is used to create the relational schema where each element is represented in its own relation.

*Flexible mapping* methods build on the fixed mapping methods, by searching the set of possible mapping for the “best” possible mapping. The set of possible mappings are accomplished by using an XML-to-XML transformation to create a new schema and then comparing the new schema against the previous. The possible transformations include: inlining/outlining, union factorization/distribution, repetition merge/split, wildcard rewriting and from union to options. An example of this method is the LegoDB mapping method.

*Intermediate mapping* methods seek to transform the XML schema into either an object or Enhanced Entity-Relationship structure and then mapping the resulting structure to a relational schema.

**User-Defined Mappings.** User-defined mappings are commonly employed in commercial databases and are system-dependent mappings. The designer must explicitly specify how elements are mapped to a defined relational schema.

For a more in-depth review and depictions of these mapping techniques see [MP05].

### Maintaining Identity, Structure and Order

XML is inherently ordered and has a document structure. The relational data model is set based and therefore does not reflect the ordered nature of the XML data model. However, various techniques may be employed to compensate for these shortcomings. Use of the foreign key references between parent and child maintain the overall structure. Order may be encoded in a number of ways. A global numbering of nodes may be created, a local numbering may be adopted to maintain order among siblings or a further “dewey” encoding may be employed. Dewey encoding stores a unique vector that describes the path from the root to the node. It is similar to a XPath expression, but is based on the ordering of siblings at each step. A global numbering provides the greatest efficiency for stable data that is often queried. A local numbering allows for an increased updating efficiency. The dewey encoding technique provides a balanced approach that fits a data set that is both queried and updated.

For a more in-depth review of the labeling techniques see [XLWB09].

### 4.2.3 Storage Overview

In this section several approaches for storing XML documents in a database were discussed. NXDs provides flexibility and freedom from mapping when storing well-formed XML documents. RDBMSs may or may not provide an XML data type or data type capable of storing an entire XML document, such as the CLOB or BLOB. The XML document may otherwise be mapped to standard relations or tables. The three main categories for mapping XML documents are: generic, schema-based and user-defined methods. The document structure may also be maintained using child/parent foreign keys, sibling-ordinal values, interval numbering, dewey number as well as a number of other techniques. Figure: 4.2 depicts a tree describing the relationship between the various techniques for storing XML documents in databases.

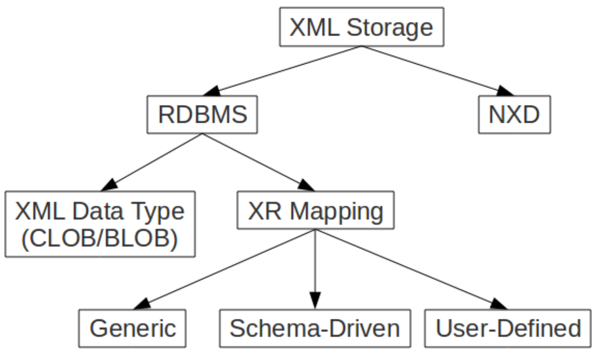


**Figure 4.2:** XML Storage Overview

## 4.3 Database Deployment Techniques

Databases may be employed in a wide variety of methods, from an embedded database to a database server to a cluster of distributed databases. However, due to the expected database usage profile for the .getmore system and the scope of requirements, only the embedded database and database server will be discussed here.

### 4.3.1 Embedded Database

Embedded Database is used to describe the use of a database that is tightly integrated into the application software. The embedded database is loaded into and shares the same thread and memory as the application using the database. This is typically accomplished by including the database executable files into the application and directly accessing the database's functionality though calls to the respective database's APIs. This is a lightweight approach where the application has complete control over the database it uses. However, since this technique embeds the database into the application, other applications are not able to remotely access the database.

### 4.3.2 Database Server

A database employed as a server is a separate application that is started and uses its own thread or process and memory. It provides access either locally or remotely through the use of an appropriate API to another applications. This allows multiple application clients to access a single database server and manipulate a communal set of data. This approach is suitable for distributed applications requiring remote access or replication. However, this approach has a larger footprint and requires more resources, as it is a separate application.

The embedded database provides a lightweight solution for a single application accessing data persisted in a database and shares the same process and memory. The database server is a separate application, which provides database services to other applications either locally or remotely. Figure: 4.3 is a visualization of the difference between the two employment techniques.

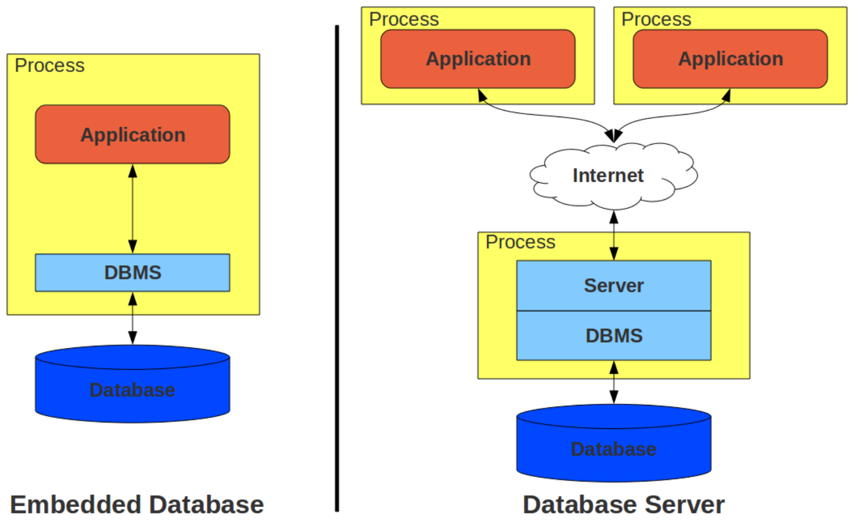


**Figure 4.3:** Embedded Database and Database Server

## 4.4 Database Comparison

Four databases, that were determined to be likely candidates for use in this project, were compared. The compared databases are: BaseX, eXist-db, Sedna and MySQL. The fundamental requirements that must be met for the database to be considered were that it was available through a non-proprietary license and must be well supported or experiencing continued development. The following is a summary of the compared database and a decision support matrix for comparing the databases.

### 4.4.1 BaseX

BaseX is a lightweight open source NXD implementation, which is an offspring of the Database and Information Systems (DBIS) Group at the University of Konstanz. It is a compact, high-performance database with very high compliance with the W3C XPath and XQuery Recommendations. It may be employed as either an embedded or database server and provides transactional support. It is developed in Java, requiring at least Java 1.6, and supports access through the Unified Representational State Transfer Access to XML Resources (REST/JAX-RX), XQuery for Java (XQJ) and XML Database API (XML:DB API), which is an initiative for XML databases. Its latest release was version 6.3.2 on 17, November 2010. It has good documentation, is actively being developed and is available for use under the Berkley Software Distribution (BSD) license [Grü10].

### 4.4.2 eXist-db

The eXist-db is an open source NXD implementation, which was first developed by Wolfgang Meier in 2000. It uses the XML data model to store data and offers efficient index-based XQuery processing. It supports the use of XQuery 1.0, XPath 2.0 and Extensible Stylesheet Language Transformations (XSLT) 1.0 and 2.0 depending on the supporting processor chosen. It also provides support for XUpdate and XQuery Update Extensions for the direct manipulation of persisted XML data. It may be employed as an embedded database or database server and is accessible through a number of interfaces to include: XML:DB API, Representational State Transfer (REST), Web-based Distributed Authoring and Versioning (WebDAV), Simple Object Access Protocol (SOAP), XML-

Remote-Procedure-Call (XML-RPC) and the Atom Publishing Protocol. It is developed in Java, requiring at least Java 1.5, and its latest release was version 1.4 in November 2009. It has good documentation, is being actively developed and available for use under the GNU's Not Unix Lesser General Public License (GNU LGPL) [eXi10].

### 4.4.3 Sedna

Sedna is an open source NXD implementation, which is supported by the Russian Academy of Science. It supports the use of XQuery and employs a declarative node-level update language to directly manipulate the stored XML data. It also provides transactional support, indices, hot backups and fine-grained XML triggers. It may be employed as an embedded database or as a database server allowing access through the XML:DB API or its own binary protocol. It is implemented in C/C++ and provides driver support for most common languages, including Java. Its latest release was version 3.3 in March 2010. It has sufficient documentation, is being actively developed and is available for use under the Apache License 2.0 [Sed10].

### 4.4.4 MySQL

MySQL is a well know and widely used RDBMS, originally being released in May 23, 1995 and has experienced continuous support and development since. Currently it is owned and sponsored by MySQL AB, which in turn is owned by Oracle Corporation. It stores XML as a BLOB and provides query (similar to XPath 1.0) and update functionality to directly manipulate the XML data. It may be employed as a database server and is typically accessed through a Java Database Connectivity (JDBC) driver. It provides the transactional support and stability associated with a grounded RDBMS. It is implemented in C/C++ and its latest release, version 5.5, was announced on 19 September 2010. It has sufficient documentation, is being actively developed and is available for use under the GNU's Not Unix General Public License (GNU GPL)[MyS10].

### 4.4.5 Decision Support Matrix

Each of these databases was subjectively and relatively rated in multiple categories such as the database type, implementation language, XML technology support, stability, access possibilities and transactional support. A lower rating number is better. The best database implementation receives a one in the category and the other databases receive a relatively increasing number. If multiple databases are perceived to have a similar quality in a category they may receive the same number. The categories were then weighted according to their relative importance, with more important categories receiving a greater weight. The XML structures to be stored are not stable and are expected to fluctuate in the future. This encouraged the use of a NXD, which provided the needed flexibility for this adaptation. The .getmore system is developed in Java, therefore a Java implementation was desired. The support for XML technologies such as XPath and XQuery also provided additional weight to support future developments. In order to determine which database implementation receives the best rating, the number in each category is multiplied by its weighting factors and are then summed. The database with the lowest overall sum is the relatively better database. Table: 4.1 depicts the decision support matrix resulting from the comparison of the four databases.

The eXist-db was determined to be the most appropriate database for use during this project. It is an open source NXD with very good support for key XML technologies. It may be employed as an embedded database or as a database server. It has a flexible modularly extensible architecture and is implemented in Java. However, it does not provide high-level access to transactional support. This was determined not to be a significant risk, because the database will be primarily employed for a small number of users accessing the database primarily through read operations.

| Criteria | wgt | **BaseX** | **eXist-db** | **Sedna** | **MySQL** |
|---|---|---|---|---|---|
| **DB Type** | **4** | 1 | 1 | 1 | 2 |
| **XML Spt** | **3** | 2 | 1 | 3 | 4 |
| **Language** | **2** | 1 | 1 | 2 | 2 |
| **Stability** | **1** | 2 | 2 | 2 | 1 |
| **Access** | **1** | 2 | 1 | 3 | 3 |
| **Trans Spt** | **1** | 1 | 2 | 1 | 1 |
| **Other** | **1** | 2 | 1 | 3 | 2 |
| Result: | | **19** | **15** | **26** | **31** |

**Table 4.1:** Decision Support Matrix (Lowest Result is Best)

## 4.5 Summary

In this chapter XML and related technologies were first discussed. Attention was paid to the parsing and querying of XML data. XML may be parsed either through the SAX or StAX streaming APIs or through the DOM tree-treversal API. XML data may be queried through the use of XPath or XQuery languages. Next the storage of XML in databases and database employment possibilities were discussed. XML data may either be mapped or stored in a specialized XML data structure within an RDBMS or it may be directly stored in a NXD. Databases may be employed in numerous different configurations, but only embedded databases and database servers were discussed. These are the primary employment methods expected to be encountered during this project.

Next four different candidate databases were introduced, described and compared. BaseX, eXist-db and Sedna are NXDs while MySQL is a RDBMS. BaseX and eXist-db are implemented in Java while Sedna and MySQL are implemented in C/C++. BaseX and eXist-db have significant support for various W3C XML standards while Sedna and MySQL do not emphasize their standards compliance. All are available for use under an open source license. The eXist-db was selected as the database implementation to use because it is a NXD, allowing XML Schema flexibility, and supports the majority of XML technologies and methods of access.

# 5 Legacy System Analysis

In this chapter the .getmore system will be analyzed. First an analysis of the overall .getmore system, focusing on the definition of the system's primary subsystems, will be provided. Next an in-depth analysis of how the internal data structures of the .getmore system are persisted and retrieved is presented. Then how the .getmore's internal data structures may be loaded into or dumped directly from the persistence subsystem is analyzed. Finally, the current persistence implementation's coupling to the `java.io.File` class will be discussed.

## 5.1 .getmore Primary Subsystems

The primary components are the .getmore Core, Persistence and Graphical User Interface (GUI) subsystems. The Core is responsible for the basic functionality of the system and generates the test cases. The Persistence Layer is responsible for storing and retrieving the internal data structures used by the Core. The GUI provides a graphical interface for the human user to interact with the system. Figure: 5.1 on the following page depicts the three primary .getmore components and their general relationship.

### 5.1.1 The Core Subsystem

The .getmore Core is responsible for importing a representation of the System Under Test (SUT), generating test cases for the SUT and exporting the generated test cases to a testing environment. The SUT to be imported may be described through a UML activity, state or sequence diagram. The imported SUT is internally referred to as the Model, which is described through a set of nodes, called vertexes, and a set of directed edges connecting the nodes. The combined nodes and directed edges result in a directed edge graph (DG), which describes the system's flow of control. Each node and directed edge

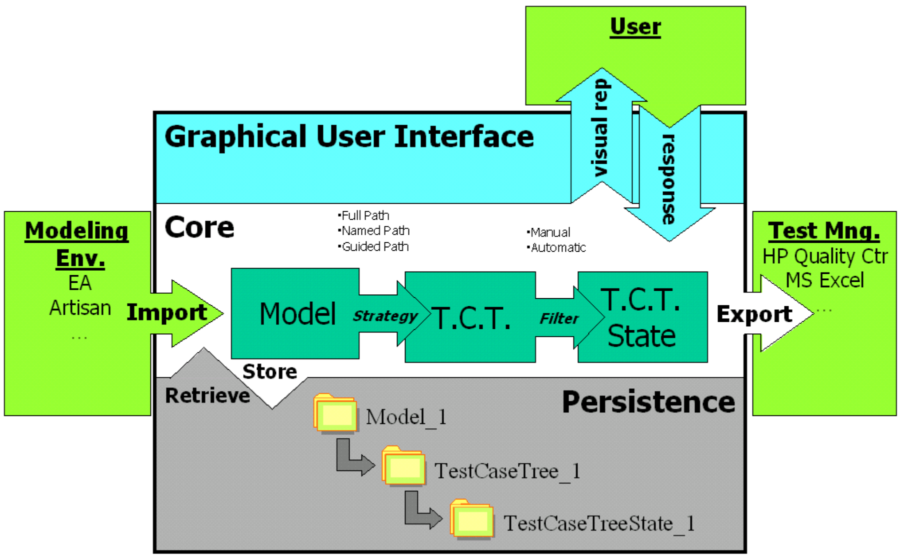


**Figure 5.1:** .getmore Primary Subsytems

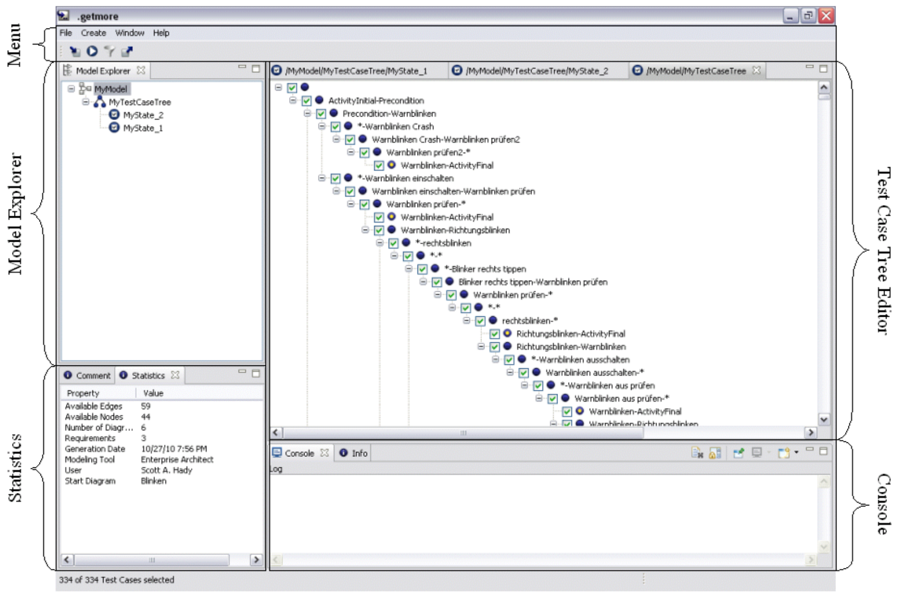
is then parameterized, according to the information presented in the UML diagram. The nodes are parameterized with implementation details such as the actions they represent. The edges are parameterized with their priority, weight, transition type and any guard conditions.

The Core applies a testing strategy to the Model in order to automatically generate a tree of test cases that fulfill the designated testing strategy's criteria. The root of the tree is always the node identified as the start node in the diagram. The available testing strategies include: Full Path Coverage (FPC), Named Path (NP), Guided Path (GP) and Random. To prevent infinite loops within the test cases, a maximum path length and number of loops must be provided. FPC attempts to cover each of the possible paths through the modeled diagram. NP tests a path, which has been predefined in the diagram. GP extends the NP strategy to multiple separately defined paths and provides the ability to test all possible combinations of the paths. The random strategy generates a path through the diagram by non-deterministically choosing the next edge along which to traverse. Even though this is a random strategy the results are repeatable for a given model.

The complete set of test cases, which are represented through the Test Case Tree, exhibits significant redundancy. The complete set of test cases may then be filtered either manually or automatically in order to eliminate waste. The user may manually filter the Test Case Tree by selecting or deselecting desired test cases. Automatic filtering algorithms are broken into two categories: coverage and range filters. Coverage filters include: node, edge, verification point and requirements. Coverage filters ensure that the defined filtering items are covered at least once during the execution of the chosen test cases. Range filters include: cost, duration and length. Range filters create a minimalist set of test cases for which the cumulative sum does not exceed a specified value.

## 5.1.2 The GUI Subsystem

The Graphical User Interface (GUI) supports the interaction between the user and the system, allowing the user to control the flow and execution of the system, and the visual representation of .getmore's data structures. It is these visualizations that the user may manipulate to manually filter the Test Case Trees. Figure: 5.2 is a screen shot of the .getmore system's GUI.



```
MyModel
  MyTestCaseTree
    MyState_2
    MyState_1
```

/MyModel/MyTestCaseTree/MyState_1 | /MyModel/MyTestCaseTree/MyState_2 | /MyModel/MyTestCaseTree

```
ActivityInitial-Precondition
  Precondition-Warnblinken
    *-Warnblinken Crash
      Warnblinken Crash-Warnblinken prüfen2
        Warnblinken prüfen2-*
          Warnblinken-ActivityFinal
    *-Warnblinken einschalten
      Warnblinken einschalten-Warnblinken prüfen
        Warnblinken prüfen-*
          Warnblinken-ActivityFinal
          Warnblinken-Richtungsblinken
            *-rechtsblinken
              *-*
                *-Blinker rechts tippen
                  Blinker rechts tippen-Warnblinken prüfen
                    Warnblinken prüfen-*
                      *-*
                        rechtsblinken-*
                          Richtungsblinken-ActivityFinal
                          Richtungsblinken-Warnblinken
                            *-Warnblinken ausschalten
                              Warnblinken ausschalten-*
                                *-Warnblinken aus prüfen
                                  Warnblinken aus prüfen-*
                                    Warnblinken-ActivityFinal
                                    Warnblinken-Richtungsblinken
```

| Property | Value |
|---|---|
| Available Edges | 59 |
| Available Nodes | 44 |
| Number of Diagr... | 6 |
| Requirements | 3 |
| Generation Date | 10/27/10 7:56 PM |
| Modeling Tool | Enterprise Architect |
| User | Scott A. Hady |
| Start Diagram | Blinken |

334 of 334 Test Cases selected

**Figure 5.2:** .getmore GUI Screenshot

The GUI is broken into five components: the Model explorer, Test Case Tree editor, statistics, console and the menu/toolbar. The Model explorer displays the system's Models that have been imported into the system as well as any respective Test Case Trees and Test Case Tree States that have been produced. The elements may be selected, renamed, expanded or collapsed and protected or unprotected. Protecting a Model means that it may not be renamed or deleted. The statistics component provides information about how the Model, Test Case Tree or Test Case Tree State selected in the Model explorer was created. The Test Case Tree editor displays the Test Case Tree or Test Case Tree State selected in the Model explorer. Each node and edge is represented as an entry in the editor, which then may be manually selected or deselected to create a desired state. Additionally, each node and may be expanded or collapsed to enhance visibility of the entire Test Case Tree. The console provides logging feedback from the system to the user.

### 5.1.3 The Persistence Subsystem

The persistence subsystem is responsible for persisting the internal data structures of the .getmore Core and on demand returning these internal data structures in the same state as which they were persisted. The details of the persistence subsystem will be described in the next section.

## 5.2 Persistence of .getmore's Internal Data Structures

The persisted data can be broken into two separate concerns: system data and user data. The system data is defined as the data explicitly required for the functionality of the .getmore system. This includes the Model, Test Case Tree, Test Case Tree State, version, locks and the element's respective metadata. The user data consists of the data that is only relevant to the display of the data in the GUI's Model explorer, the expanded file. For a single user system, the two concerns may be merged. However, for a multiple user system, these concerns must be separated so the preferences of each individual user can be distinguished.

Persistence is accomplished in the persistence layer through the use of Data Access Objects (DAOs), which correspond to the internal data structures used by the .getmore Core. The DAO is identified through their Info suffix appended to the respective data structure name. The DAOs follow an atypical implementation. The storing and retrieving of an internal structure is not accomplished by the same DAO. The DAO responsible for storing a structure is also responsible for returning an array of DAOs that may then be used to retrieve the element's subelements. Figure: 5.3 on the facing page depicts the key DAO classes used in the persistence subsystem so the relationship between the DAOs and internal data structures can be visualized.

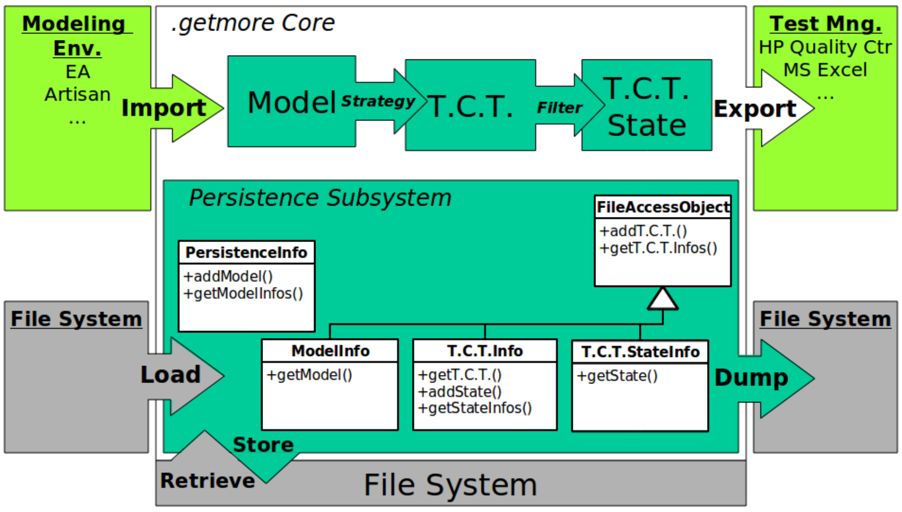


**Figure 5.3:** Persistence Subsystem Data Access Objects

Each of the ModelInfo, TestCaseTreeInfo and TestCaseTreeStateInfo classes inherit the functionality provided by the FileAccessObject. The FileAccessObject encapsulated a common set of functionality provided by each of the DAOs. This functionality includes renaming the structure, getting and setting the metadata and removing the element. Additionally, the FileAccessObject provides functionality to add a Test Case Tree and return the respective DAOs used to access the Test Case Trees. This is an interesting design decision because only the Model element may contain a Test Case Tree, but the functionality is accessible to each element.

The functionality of the persistence subsystem's primary classes is described through a set of interfaces, communally known as the persistence layer's interface. The interfaces represent a specification or contractual agreement, describing how the subsystem may be accessed and used by a client. This is known as Programming-to-Interfaces [GHJV95]. It provides a decoupling mechanism between subsystems' implementations, allowing them to be implemented independently from each other. Programming-to-Interfaces also makes developing a new implementation easier because the client or using subsystem is not directly aware of the implementation details used to fulfill the interface's contract. In the .getmore system, interface classes are predicated with an "I" to facilitate their easy identification. Figure: 5.4 on the next page provides a depiction of the primary classes and subcomponents of the persistence subsystem. The File System, XML Processing, Converter and Dumper are all components commonly used by all classes within the persistence layer. Likewise, the Message class is used to retrieve designated messages for logging and exception handling and commonly used by all classes.

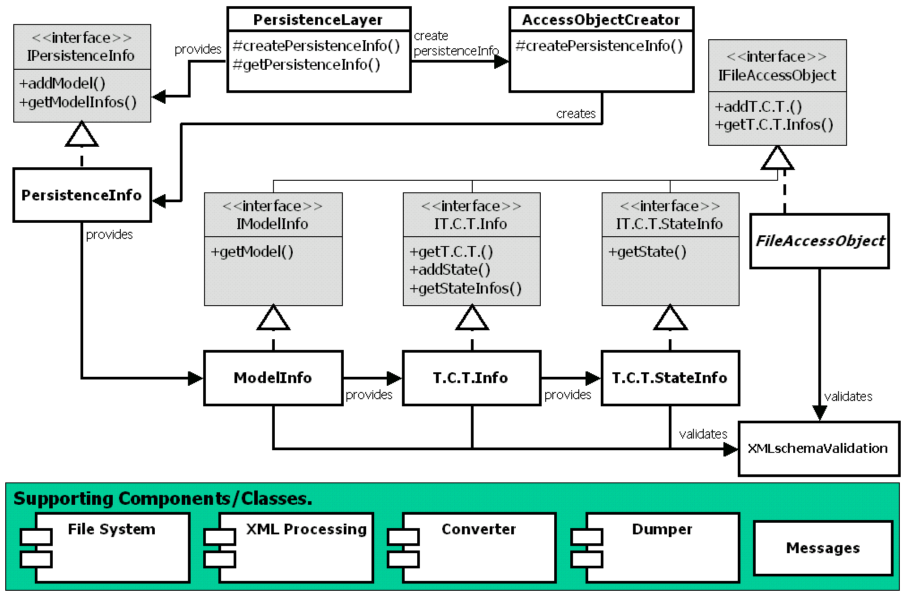


**Figure 5.4:** Persistence Subsystem Class Diagram

The .getmore Core accesses the persistence subsystem by requesting an instance of the PersistenceInfo class from the PersistenceLayer class. The PersistenceInfo is typically treated as a singleton within the .getmore system and its access is similar to the Singleton design pattern. However, the creation of the PersistenceInfo must first be explicitly requested through a static call to the PersistenceLayer and it is not guaranteed that only a single instance will be created by the system. When the creation of a new PersistenceInfo object is requested, the PersistenceLayer will forward the request to the AccessObjectCreator. The AccessObjectCreator is responsible for the creation of the new PersistenceInfo object. It will create the new object and return it to the PersistenceLayer. The PersistenceLayer maintains the returned instance and provides it to the .getmore Core as needed. The .getmore Core also provides instances of the Model, Test Case Tree and Test Case Tree State factories when it requests the creation of a new PersistenceInfo Object. These factories are then used by the subsystem in order to create the objects retrieved from the file system.

### 5.2.1 Storing Internal Data Structures

Persisting the .getmore's internal structures follows a relatively simple pattern of persisting an element and then using the returned DAO in order to persist its subelement. Figure: 5.5 depicts the basic activities for persisting a Model with a Test Case Tree and a Test Case Tree State. First an instance of IPersistenceInfo must be acquired. IPersistenceInfo provides the functionality to persist a Model and an instance of the IModelInfo DAO is returned. The IModelInfo may be used to store a Test Case Tree, which in-turn returns a ITestCaseTreeInfo. A Test Case Tree State may then be persisted using the functionality provided by the ITestCaseTreeInfo.

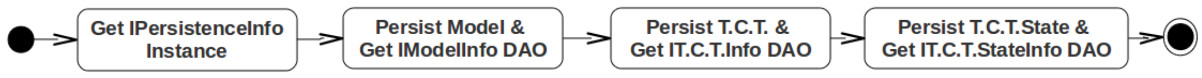

Get IPersistenceInfo Instance
Persist Model & Get IModelInfo DAO
Persist T.C.T. & Get IT.C.T.Info DAO
Persist T.C.T.State & Get IT.C.T.StateInfo DAO

**Figure 5.5:** Storing Internal Data Structures Overview

#### 5.2.1.1 Persisting Models

Once the .getmore Core has an instance of PersistenceInfo, it may add a Model to persistence. First the PersistenceInfo creates a subdirectory in the persistence directory with given Model's name. Then the PersistenceInfo uses the ModelFileCreator, within the XML processing subsystem, to create a DOM representation of the Model and serializes the representation to a .model XML file. Once the Model file is created a ModelInfo instance is created and the file is validated. The Model's metadata is likewise represented as a DOM and serialized to a .metadata XML file within the Model's subdirectory, through the use of the XML processing subsystem's MetaInfoFileCreator. The PersistenceLayer is then requested to create a .version file which identifies the .getmore system for which this Model was created. After the Model's information has been stored, the PersistenceInfo returns an instance of ModelInfo to the .getmore Core.

Validation is decoupled from parsing using the Java API for XML Processing (JAXP) Validation API is accomplished when the respective DAOs are created. The ModelInfo validates the Model files, the TestCaseTreeInfo validates the .tct files, the TestCaseTreeStateInfo validates the .state files and the FileAccessObject validates the .metadata files. Validation for the primary elements is executed when the elements are stored and retrieved. However, the .metadata file is only validated when it is retrieved from the FileAccessObject because the .metadata file is not created until after the class' instantiation. Figure: 5.6 is a diagram that provides a visual depiction of the sequence of events that occur when the .getmore Core adds a Model to the persistence subsystem. This is a visual illustration of the previous discussion.

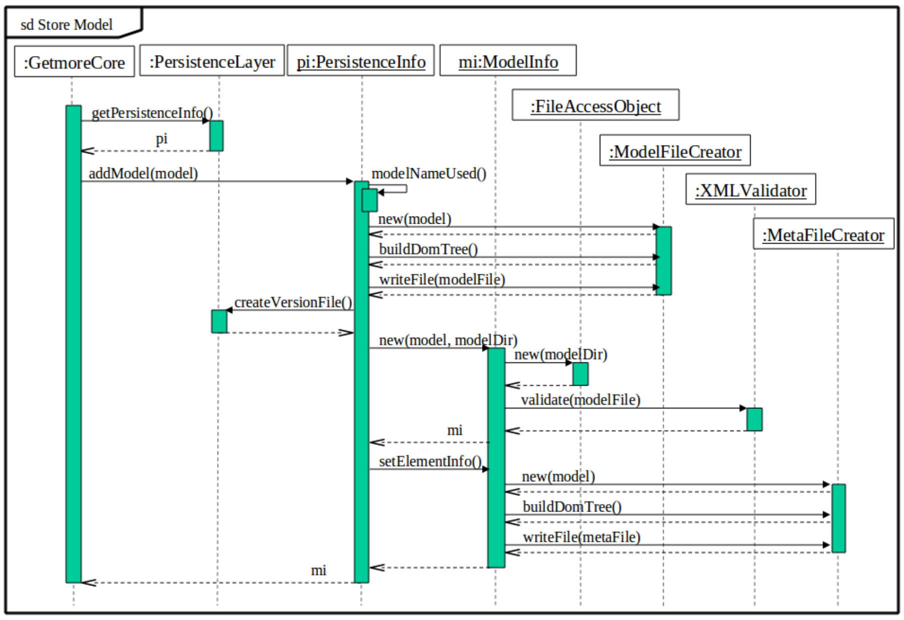


**Figure 5.6:** Model Storage Sequence Diagram

#### 5.2.1.2 Persisting Test Case Trees

Once the .getmore Core has an instance of ModelInfo, it may use it to add a Test Case Tree generated for the respective Model to persistence. First the ModelInfo creates a subdirectory named after the Test Case Tree. The ModelInfo also uses the TCTFileCreator and MetaInfoFileCreator, within the XML processing subsystem, to first generate a DOM representation of the Test Case Tree and metadata and then serializes the representations to the .tct and .metadata XML files within the Test Case Tree's subdirectory. Once the Test Case Tree's information has been stored, the ModelInfo returns an instance of the TestCaseTreeInfo to the .getmore Core. It is important to note that the functionality used by the ModelInfo is actually implemented in and inherited from the FileAccessObject class.

#### 5.2.1.3 Persisting Test Case Tree States

Once the .getmore Core has an instance of the TestCaseTreeInfo, it may use it to add a state of the Test Case Tree to persistence. The TestCaseTreeInfo first creates a subdirectory for the Test Case Tree's state. It then uses the StateFileCreator and MetaInfoFileCreator, in the XML processing subsystem, to first generate a DOM representation of the Test Case Tree State and metadata and serializes the representation to a .state and .metadata XML file within the state's subdirectory. Once the Test Case Tree State has been stored an instance of the TestCaseTreeStateInfo is returned to the .getmore Core.

### 5.2.2 Retrieving Internal Data Structures

Retrieving the .getmore's internal structures follows a relatively simple pattern of using an element's DAO to retrieve the element and to retrieve the DAOs for its subelements. First an instance of IPersistenceInfo must be acquired. IPersistenceInfo provides the functionality to get all of the Model DAOs which are capable of retrieving the Models and the DAOs for their Test Case Trees. A similar recursive pattern is used to retrieve the persisted Test Case Trees and Test Case Tree States. Figure: 5.7 depicts the basic activities for retrieving all persisted Models, Test Case Trees and Test Case Tree States.

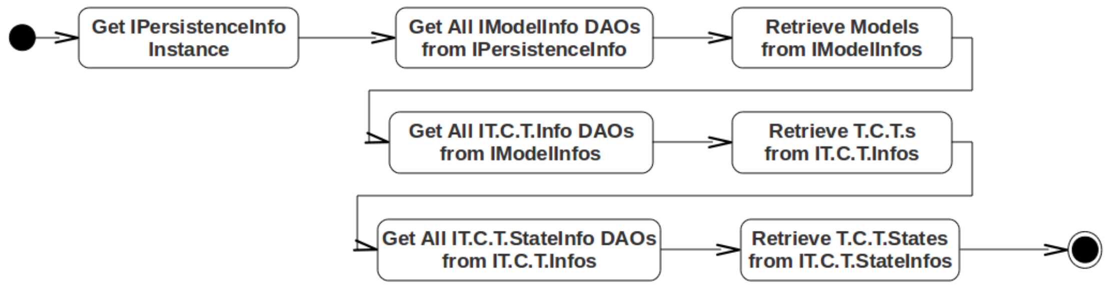

Get IPersistenceInfo Instance
Get All IModelInfo DAOs from IPersistenceInfo
Retrieve Models from IModelInfos
Get All IT.C.T.Info DAOs from IModelInfos
Retrieve T.C.T.s from IT.C.T.Infos
Get All IT.C.T.StateInfo DAOs from IT.C.T.Infos
Retrieve T.C.T.States from IT.C.T.StateInfos

**Figure 5.7:** Retrieving Internal Data Structures Activity Diagram

An element's DAO is returned by the persistence subsystem when the element is persisted or it may be returned as part of the set provided by the #getXXXInfos method of the superior element's DAO. Each DAO is parameterized with the directory in which its respective element is stored and the element's name and the element's base Model if it is a Test Case Tree or Test Case Tree State.

The .getmore Core acquires a persisted Model by requesting its retrieval from the Model's corresponding ModelInfo DAO. First the ModelInfo determines if the Model has already been retrieved an available to be returned. If the Model has not been retrieved it must parsed from the stored `.model` XML file. A SAXEventHandler and a ModelReader, both defined in the XML processing subsystem, are first instantiated. The ModelReader creates a Model instance from the registered PersistenceLayer's Model factory and is registered as a listener for the SAX events. When the file is parsed, the ModelReader receives the requested event notifications and populates the Model object. After the file has been parsed the Model is retrieved by the ModelInfo instance, which then returns the Model to the .getmore Core. The Model's metadata is similarly retrieved through the functionality inherited the FileAccessObject. The .getmore Core acquires a persisted Test Case Tree or Test Case Tree State in a very similar process through their respective TestCaseTreeInfo or TestCaseTreeStateInfo. Figure: 5.8 is a sequence diagram that depicts the sequence of events that occur when the .getmore Core retrieves the persisted Models from the persistence subsystem. This is a visual representation of the previous discussion.

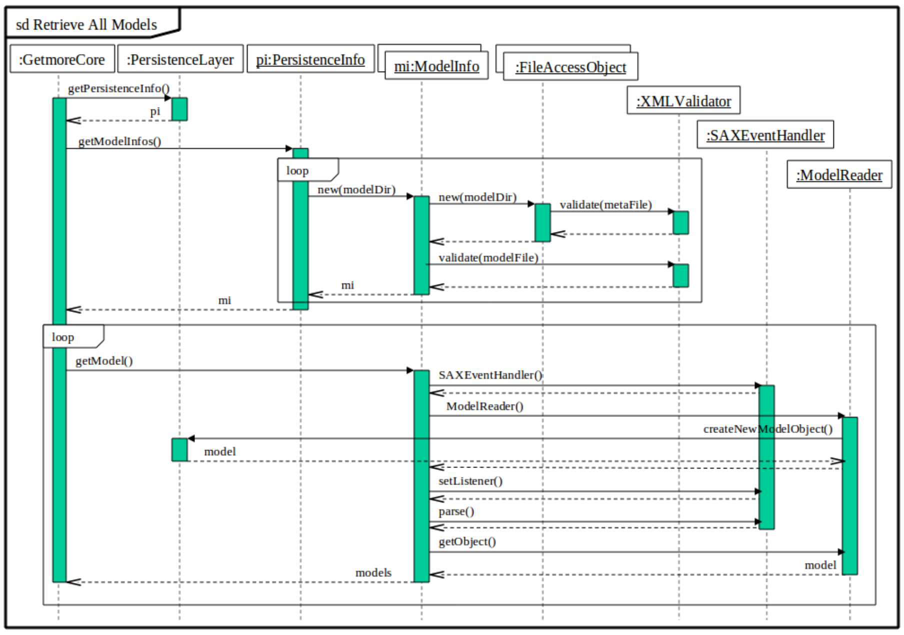


**Figure 5.8:** Model Retrieval Sequence Diagram

## 5.3 Loading, Converting and Dumping Persisted Data

In the .getmore system the internal data structures are “dumped” from or “loaded” into the persistence subsystem in order to provide a simple method of transferring data from one .getmore application to another, backup and restore a set of Models or to allow a user to open and visually inspect the XML documents. The terms “dump” and “load” are used because the terms “import” and “export” are used to refer to the importing of Models from a modeling environment and the exporting of the generated test cases to a test management system. Dumping refers to the task of generating a copy of a Model in the persistence subsystem in a file system directory. Loading refers to the task of generating a Model in the persistence subsystem from the contents of a file system directory. However, this terminology was not established until after the file system persistence subsystem was created. Therefore, the terms “import” and “export” are ambiguously used in the legacy system. Conversion is the process of rejuvenating a Model imported by an older version of .getmore into a format expected by a newer version of .getmore. This allows the user to continue using a set of data while upgrading the .getmore system to a new version.

### 5.3.1 Dumping Persisted Data to the File System

The dumping of the persisted elements into a designated file system directory is a simple task. First the user designates a target directory where the persisted data should be dumped. Then the user designates which Models to dump. Since the Models are persisted as files in the files system, the dumping of the Model is a simple copy of the persisted Model’s directory into the designated directory.

### 5.3.2 Loading Data from the File System into Persistence

The basic loading of a Model directly into the persistence subsystem is also a relatively simple process of copying the designated directory into the persistence subdirectory and notifying the system of the new Models. However, two anomalies may occur and must be dealt with. First, since the Models are identified by their name, which is not guaranteed to be unique, it may occur that the system attempts to load multiple Models with the same name. In order to prevent a naming collision, the user must choose an appropriate handling method from a set of radio buttons provided by the GUI. The user may choose to ignore the new Model and keep the previously persisted Model, overwrite the persisted Model with the new Model or modify the Model's name to avoid the conflict. Second, the Model may have been created for a different version of the .getmore system and must be converted to an appropriate format.

### 5.3.3 Converting Outdated Models

Conversion modifies a Model created by an older version of the .getmore system into a format supported by the current version of the .getmore system. First the `.version` file of the Model is reviewed to determine the version of .getmore for which it was created. If the version is the same as the current .getmore's version nothing is done. If the version is newer than the current .getmore system, the Model is archived and must be manually converted prior to being loaded. If the Model's version is older than the current system's version, it is annotated and moved to a designated restoration directory. Next the converter's configuration is parameterized with the location of the restoration directory, the persistence directory and the method for handling naming conflicts. Finally the converter manager is requested to execute the conversion. The conversion manager uses the set configuration details and the appropriate converter, based on the current .getmore system, in order to convert the Models.

### 5.3.4 Loading Models into .getmore

Figure: 5.9 on the facing page is sequence diagram that depicts the sequence of events that occur when new Models are loaded into the persistence subsystem. This is a visual representation of the previous discussion. The ConverterAction represents the overall conversion process. The ConverterDialog is responsible for retrieving the user's input through use of the ConverterComposite GUI and setting the appropriate settings in the ConverterConfiguration. If ConverterDialog return designates that there are Models in need of conversion, the ConverterAction will request that the ConverterManager convert the Models. The ConverterManager retrieves the data directory and selected Models from the ConverterConfiguration. It will then attempt to convert any Models that are determined to be of the incorrect version. Once all Models are converted, the RootFolderProxy is notified in order to update the GUI's Model explorer.

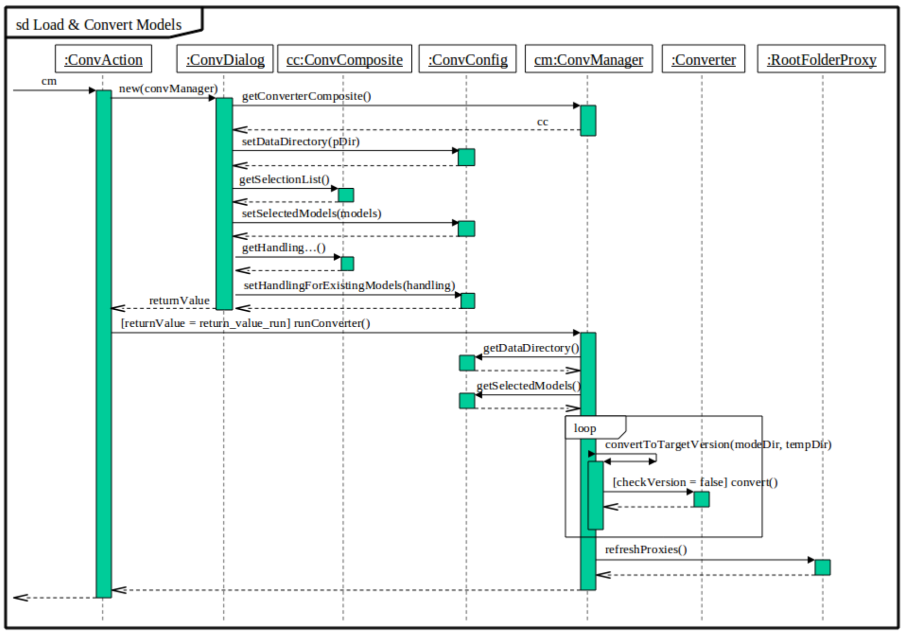
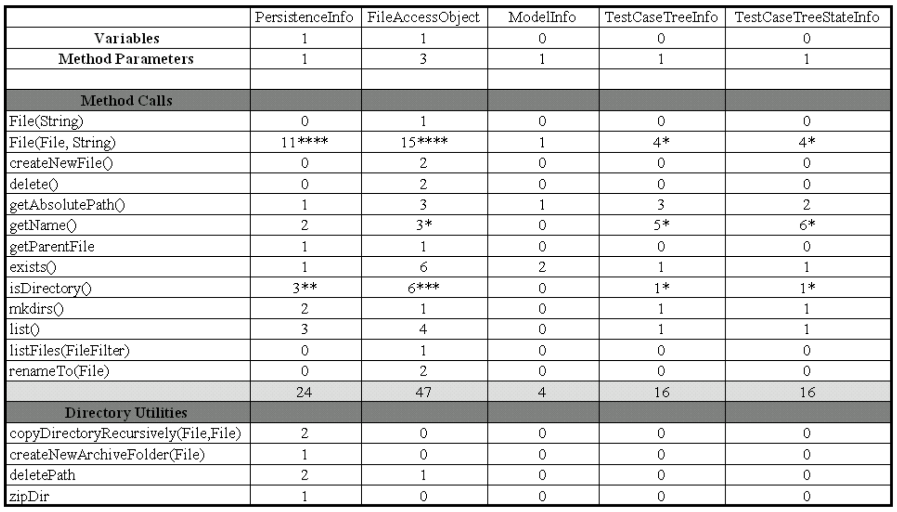


**Figure 5.9:** Model Loading and Conversion Sequence Diagram

Every time the .getmore system is started, all persisted Model's versions are checked and outdated Models are converted in a similar manner in which they are converted prior to being loaded into the persistence subsystem.

## 5.4 Dependency on java.io.File

The persistence subsystem depends heavily on the use of the java.io.File class. Calls directly to the java.io.File class occur at a high rate within the subsystem, which is a symptom of high coupling between the subsystem and the java.io.File class. 107 calls are made to the File class from the five primary classes of the file system persistence implementation. The File class is also directly referred to by the persistence layer's interface and is used in the signature of seven of its methods. The DirectoryUtilities is a class that provides static helper methods for file system manipulation using the File class. Finally, the XML processing component also uses the File class directly. The majority of the parsing may be used in a manner that avoids its use. However, the XMLschemaValidation class, which is part of the getmoreSysLib package, only provides a single validation method. This method depends on the File class in a manner in which it may not be avoided. Table: 5.1 depicts the high level of the coupling between the file system persistence subsystem's implementation and the java.io.File class. Each asterisk annotates an iterative call to the class. These are specially noted because the exact number of calls made cannot be statically determined.



| | PersistenceInfo | FileAccessObject | ModelInfo | TestCaseTreeInfo | TestCaseTreeStateInfo |
|---|---|---|---|---|---|
| **Variables** | 1 | 1 | 0 | 0 | 0 |
| **Method Parameters** | 1 | 3 | 1 | 1 | 1 |
| | | | | | |
| **Method Calls** | | | | | |
| File(String) | 0 | 1 | 0 | 0 | 0 |
| File(File, String) | 11**** | 15**** | 1 | 4* | 4* |
| createNewFile() | 0 | 2 | 0 | 0 | 0 |
| delete() | 0 | 2 | 0 | 0 | 0 |
| getAbsolutePath() | 1 | 3 | 1 | 3 | 2 |
| getName() | 2 | 3* | 0 | 5* | 6* |
| getParentFile | 1 | 1 | 0 | 0 | 0 |
| exists() | 1 | 6 | 2 | 1 | 1 |
| isDirectory() | 3** | 6*** | 0 | 1* | 1* |
| mkdirs() | 2 | 1 | 0 | 1 | 1 |
| list() | 3 | 4 | 0 | 1 | 1 |
| listFiles(FileFilter) | 0 | 1 | 0 | 0 | 0 |
| renameTo(File) | 0 | 2 | 0 | 0 | 0 |
| | 24 | 47 | 4 | 16 | 16 |
| **Directory Utilities** | | | | | |
| copyDirectoryRecursively(File,File) | 2 | 0 | 0 | 0 | 0 |
| createNewArchiveFolder(File) | 1 | 0 | 0 | 0 | 0 |
| deletePath | 2 | 1 | 0 | 0 | 0 |
| zipDir | 1 | 0 | 0 | 0 | 0 |

**Table 5.1:** Persistence Subsystem and java.io.File Coupling

## 5.5 Summary

This chapter provided an analysis of the .getmore system. First the three main subsystems, the Core, GUI and Persistence Layer, were introduced and examined. Next the Persistence Layer was further investigated. The use of the persistence subsystem's DAOs by the Core subsystem in order to store and retrieve the system's internal data structures was described. Next Model loading into and dumping out of the persistence subsystem as well as the possible difficulties, naming and versioning conflicts, were described. The process of converting Models, identified to be of the incorrect version, was also described. Finally, the coupling between the persistence subsystem and the java.io.File class was accessed as high.

# 6 Development – Iteration One

This chapter will examine the developmental effort of the project's first iteration. First the agile developmental methodology applied will be described. Next the first iteration's scope, design and implementation will be explained. Finally, an assessment of the iteration's developmental effort will summarize the results.

## 6.1 An Agile Approach

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through cooperation between a team of developers, the customer and users. No specific agile method was strictly followed, rather a mix heavily influenced by Test Driven Development (TDD) [Kos08], eXtreme Programming (XP) [Bec99] and Scrum [Sch95] was applied. At the lowest level the code was developed in a test-code-refactor cycle. Code was written to make a failing test pass, this resulted in a solid product with good test coverage. 165 tests provided approximately 70% line and branch coverage. JUnit [TLMG10] was the testing framework used. However, since mock objects and stubs were not used, the tests were of a distinctly integration flavor. The developmental effort was driven at the higher levels through two iterations each of approximately one month's length, similar to Scrum's recommended sprint length. Prior to each iteration the requirements to be implemented were solidified. Once a week there was a meeting between the developer, mentor and customer in order to assess the current status of the project, consider the next week's developmental effort and clarify any requirement questions. Many values, principles and practices of XP were expressed through these efforts. For a overview of various agile and iterative approaches see [Lar04].

## 6.2 Scope of Iteration One

The first iteration's focus was the accomplishment of the project's first and primary goal: Replace the file system persistence subsystem with a database implementation. The goal is specified further through requirements one through four. Requirement eight was also designated for completion to provide a complete .getmore system suitable for distribution to the end user. See Section 3.1: *Requirements* on page 7 for a complete description of the requirements.

**R1:** Access the embedded database.

**R2:** Store the internal data structures in the embedded database.

**R3:** Retrieve the persisted internal data structures from the embedded database.

**R4:** Integrate the embedded database persistence subsystem into the .getmore system.

**R8** Integrate the database configuration into .getmore's installation process.

The end state of the product after the first iteration was defined as follows: The .getmore system persists its internal data structures, defined as the Model, Test Case Tree, Test Case Tree State and associated metadata, within an embedded database. Persistence is accomplished through use of the previously defined persistence layer interface, which is not subject to alteration. The embedded database is integrated into the .getmore system's installation process to facilitate automatic product production and installation.

## 6.3 Design – Overview

Design in the first iteration was accomplished in three phases. First, the details of the persistence layer's file system use were defined as an interface, as described in Chapter 5: *Legacy System Analysis* on page 29. Next a specification for the needed database's functionality was derived from the previously defined interface. Finally, the technical integration of the encapsulated database's functionality was completed.

### 6.3.1 Overarching Concerns

Persistence of the internal data structures must fulfill three overarching concerns.

1. **The relationship between the data structures must be maintained.** The association between a Model and its generated Test Case Trees and the association between a Test Case Tree and its filtered Test Case Tree State must be maintained in order to provide a consistent picture of the SUT.
2. **The data of each element must be maintained.** The Model's, Test Case Tree's and Test Case Tree State's data must be persisted.
   - The Model consists of a data structure that describes the imported diagram and metadata providing additional information about the Model's importation and current state of display within the Model explorer view within the GUI.
   - The Test Case Tree consists of a data structure describing the set of test cases generated through applying the designated strategy and its metadata describing the strategy's application and current state within the Model explorer.
   - The Test Case Tree State consists of a data structure describing a filtered state of the Test Case Tree as well as metadata describing the application of the filter and its current state within the Model explorer.
3. **The version of the .getmore system must be maintained.** The version of the system that first imported and last manipulated the Model must be maintained in order to ensure that it is in the correct format for any future system that attempts to access the Model.

### 6.3.2 Starting Basis

Persistence in the file system-based subsystem fulfills the first concern by creating and manipulating a hierarchical directory structure where each subelement is a subdirectory of the parent element's directory. The persistence of the elements data and metadata is accomplished through use of the XML processing subcomponent's serialization and deserialization functionality. The persistence of the element's current display state is accomplished through creating and deleting marker `.exp` and `.lock` files within the appropriate element's directory. The version of the .getmore system is maintained through the creation of a `.version` file within the Model directory. In order to accomplish all of these concerns, the file system's persistence implementation classes use the File class of the java.io package. As described in Section 5.4: *Dependency on java.io.File* on page 41, the current persistence subsystem's implementation exhibits high coupling to the File class. The persistence of the Test Case Trees and Test Case Tree States is similarly implemented.

## 6.4 Design – java.io.File Replacement

In order to replace the file system as the persistence mechanism the use of the java.io.File class must be replaced. This can be accomplished through two different approaches. The first approach is to simply replace all calls to the File class with appropriate calls to the eXist-db. The second approach is to encapsulate the eXist-db's functionality within a class and then use this encapsulation within the persistence layer's implementation.

The first approach is a relatively simple approach, but exhibits a number of serious drawbacks. First, it creates very tight coupling between the persistence layer's implementation and the database similar to that present in the current persistence layer's implementation. Second, unlike the file system, the eXist-db must be explicitly accessed prior to the use of its functionality. Such a naive approach would lead to significant access code redundancy, because each use of the databases functionality would be required to create its own access to the database. This is an example of the code duplication and the Anti-Pattern Copy and Paste Programming [BMMM98], which creates a maintenance nightmare. Finally, the persistence layer would be again directly coupled to the eXist-db. The entire implementation must be re-written if another database implementation were to be supported in the future. This is an example of the Anti-Pattern known as Vendor Lock-In [BMMM98] and should be avoided.

Encapsulating the required functionality within its own object alleviates the majority of problems identified in the first approach. It is a good use of object-oriented programming to reduce Duplication Code and maintenance efforts. The encapsulation also reduces the coupling between the persistence layer and the eXist-db. A different database implementation may also be supported as long as it fulfills the contractual agreements set by the encapsulated object. Instead of re-implementing each class, only the database encapsulation must be re-developed. Finally the control over access to the database is maintained within the encapsulation, which simplifies altering the actual access method.

Due to the aforementioned reasons, the second approach is deemed the better approach to apply.

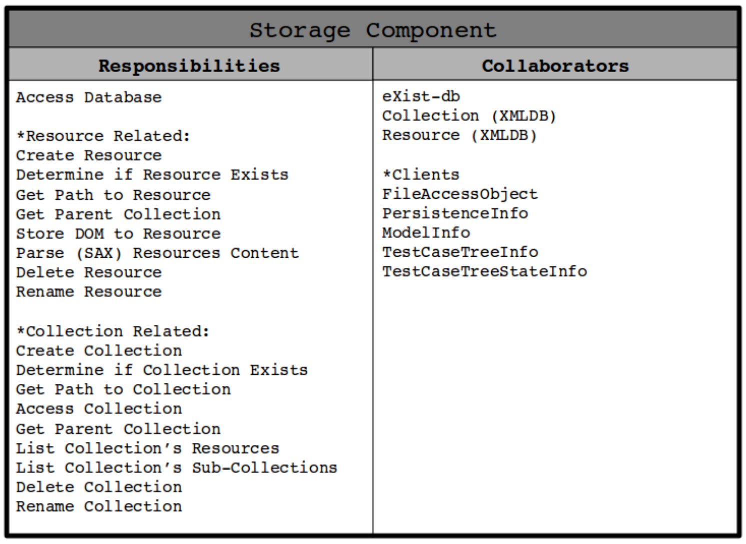
The eXist-db provides multiple interfaces through which the database may be accessed. It offers access through XML:DB API, REST, WebDAV, SOAP, XML-RPC and Atom Publishing Protocol. XML:DB API is an initiative to provide an access specification specifically for NXDs and is supported by most NXDs. The REST, WebDAV, SOAP, XML-RPC and Atom Publishing Protocols are all a Hypertext Transfer Protocol (HTTP) based specifications also supported by the eXist-db. The XML:DB API was chosen; because it is a simple vendor neutral API designed specifically for the manipulation of data within a NXD and provides the necessary functionality. Additionally, it is appropriate for use with eXist-db in embedded mode, whereas the others are based on the HTTP, which is inherently remote.

The XML:DB API specification provides a common access mechanism to a NXD, simplifying the development of applications to store, retrieve, modify and query data within a NXD [SX01]. The XML:DB API provides functionality to store and retrieve XML documents, referred to as resources. Additionally, these resources may be logically grouped into units referred to as collections. A simple parallel may be drawn between the use of this functionality and that which is used in the file system persistence subsystem. The only functionality not directly available is the ability to rename a resource or collection. However, creating a new resource or collection with the new name and copying the contents can accomplish this.

### 6.4.1 Storage Component CRC Card

The encapsulation of functionality may be provided either by a single class or a collaboration of multiple classes. Initially, due to quantity of responsibilities to be encapsulated, a collaboration of multiple classes may seem appropriate. However, the XML:DB API's Collection and Resource classes provide the actual functionality to be used. The encapsulation is actually only responsible for providing a unified interface to the appropriate API methods. This fits the intent of the Façade Design Pattern. The intent of the Façade Design Pattern is to "[p]rovide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use" [GHJV95]. Additionally, a façade does not prevent direct access to the subsystems. The subsystems may still be directly accessed and manipulated if needed. Therefore, it was determined that the encapsulation be provided through a single class which is primarily responsible for redirecting calls to the appropriate XML:DB API method. This decision is supported by the previous consideration that it would be possible to simply replace the calls made to the java.io.File class with calls to the appropriate method. However, the encapsulation will provide more functionality that the typical façade, due to the additional accessing requirements.

The responsibilities of the encapsulating class are derived from the coupling analysis between the persistence subsystem and the File class done in Section 5.4: *Dependency on java.io.File* on page 41. The File class uses abstraction to blur the distinction between a file and directory. However, since the distinction between a resource and collection is not abstracted in the XML:DB API, the majority of responsibilities must appear twice. Additionally, the collaborators can be broken into the subsystems for which the storage component provides access to and the classes that use the storage component in order to access the functionality provided by its subsystems. This level of detail is not typically associated with a Class-Responsibilities-Collaborators (CRC) card, but is important in this class and facilitates the following refinement of the class. Figure: 6.1 depicts a CRC card describing the responsibilities that must be assumed by a class in order to replace the functionality provided by the java.io.File class.



| Storage Component | |
|---|---|
| **Responsibilities** | **Collaborators** |
| Access Database<br><br>*Resource Related:<br>Create Resource<br>Determine if Resource Exists<br>Get Path to Resource<br>Get Parent Collection<br>Store DOM to Resource<br>Parse (SAX) Resources Content<br>Delete Resource<br>Rename Resource<br><br>*Collection Related:<br>Create Collection<br>Determine if Collection Exists<br>Get Path to Collection<br>Access Collection<br>Get Parent Collection<br>List Collection's Resources<br>List Collection's Sub-Collections<br>Delete Collection<br>Rename Collection | eXist-db<br>Collection (XMLDB)<br>Resource (XMLDB)<br><br>*Clients<br>FileAccessObject<br>PersistenceInfo<br>ModelInfo<br>TestCaseTreeInfo<br>TestCaseTreeStateInfo |

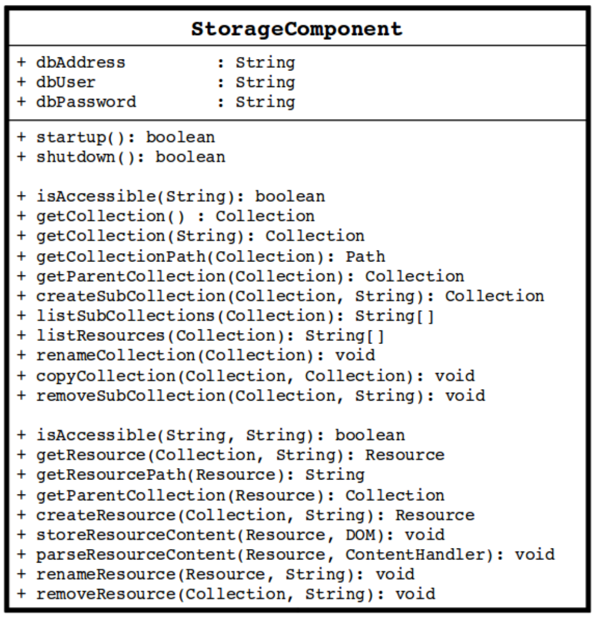
**Figure 6.1:** Storage Component CRC Card

Beyond the basic functionality provided by encapsulated subsystems there are two considerations that must be also handled by the storage component. First, the storage component will provide an access method for the database and secondly, the storage component must provide appropriate exception handling. Access to the eXist-db is a standard procedure that may be enclosed within an appropriate method of moderate complexity. Exception handling in the .getmore system must follow a specific pattern, which is not implemented in the XML:DB API. Therefore, additional error handling must support the relaying of requests.

The storage component's encapsulation reduces the coupling between the persistence subsystem implementation classes and the eXist-db. The coupling could be further reduced through the use of an interface and the Programming-to-Interfaces approach. However, this was determined not be necessary as the decoupling provided through the Façade Design Pattern was sufficient. By encapsulating the access to the eXist-db, the persistence subsystem is also insulated from the database implementation used, as long as it supports the XML:DB API. In order to change the supported database only the access methods within the StorageComponent must be changed.

## 6.4.2 Storage Component Class Diagram

Deriving a class diagram from the previously described CRC card consists of identifying the variables and methods associated with each responsibility and defining the methods' signatures. The resulting collection of method signatures is known as the classes interface. The collaborating classes then request the functionality provided through a call to the class' appropriate method, which in-turn relays the request to the appropriate subsystem. Figure: 6.2 on the next page depicts the resulting class diagram.



```
StorageComponent
+ dbAddress        : String
+ dbUser           : String
+ dbPassword       : String

+ startup(): boolean
+ shutdown(): boolean

+ isAccessible(String): boolean
+ getCollection() : Collection
+ getCollection(String): Collection
+ getCollectionPath(Collection): Path
+ getParentCollection(Collection): Collection
+ createSubCollection(Collection, String): Collection
+ listSubCollections(Collection): String[]
+ listResources(Collection): String[]
+ renameCollection(Collection): void
+ copyCollection(Collection, Collection): void
+ removeSubCollection(Collection, String): void

+ isAccessible(String, String): boolean
+ getResource(Collection, String): Resource
+ getResourcePath(Resource): String
+ getParentCollection(Resource): Collection
+ createResource(Collection, String): Resource
+ storeResourceContent(Resource, DOM): void
+ parseResourceContent(Resource, ContentHandler): void
+ renameResource(Resource, String): void
+ removeResource(Collection, String): void
```

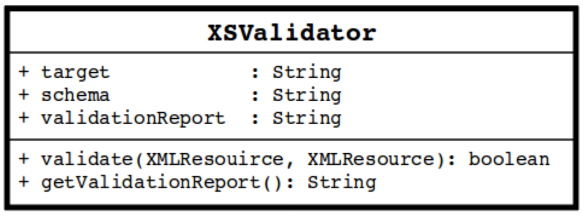
**Figure 6.2:** StorageComponent Class Diagram

## 6.5 Design – Validation Facilities

Each of the persistence system's classes must then be refactored to use the StorageComponent class instead the java.io.File class, but the majority of the current design may be assumed without change. However, as identified in the system's analysis, .getmore's validation class, XMLschemaValidation, is not suitable for use without the file system. This is because it separates the validation from the parsing and uses Java API for XML Validation (JAXV) and only provides a single method for validating XML files in the file system. There are three options to use for the implementation of the validation functionality. First, the files may be exported from the database to a designated directory and then validated. Second, the XMLschemaValidation class may be extended to also validate resources stored within the database. Lastly, the resources can be validated in the database internally using one of the validation methods provided by the eXist-db. The last option is the better option as the XMLschemaValidation is not part of the persistence subsystem, therefore best left unaltered, and better performance will be gained by validating the resources directly within the database.

The eXist-db provides support for implicit and explicit validation. Implicit validation implies that documents should be automatically validated when they are inserted into the database. Implicit validation is enabled through the eXist-db's configurations files. Within the configuration file the mode of validation and the location of the grammars to be used are defined. Documents may be explicitly validated through use of XQuery functions. XQuery extension functions for validation through JAXP, JAXV and Jing are provided. The JAXP validation functionality is based on the javax.xml.parsers API and supports the use of DTDs and XSDs grammars. JAXV validation functionality is based on the javax.xml.validation API and is intended to be independent of the grammar language. The Jing validation functionality is based on the Jing library and supports XSD, RelaxNG, Schematron and Namespace-based Validation Dispatching Language (NVDL). Beyond the true/false returned from the basic validation functionality, each of the validation frameworks provides a validation report that provides details of the last validation effort [eXi10]. In order to maintain a similar control flow, validation framework and to support a wider variety of grammars, explicit validation through the JAXV was chosen.

Based on this analysis, a new class was designed to fulfill the validation requirements within the new persistence subsystem. Figure: 6.3 depicts the XSValidator class, which is capable of validating a target resource with a designated schema resource. Additionally, the validation report is provided to assist in determining the cause invalidating a document.



```
XSValidator
+ target            : String
+ schema            : String
+ validationReport  : String
+ validate(XMLResouirce, XMLResource): boolean
+ getValidationReport(): String
```

**Figure 6.3:** XSValidator Class Diagram

## 6.6 Design – Model Conversion

Model conversion is the rejuvenation of a Model imported for an older version of .getmore to support a newer version. This allows the continued manipulation of data sets after the system has been upgraded. Conversion is directly dependant on the file system and will require further development to support the database persistence implementation.

However, for this iteration it was specifically designated as not to be altered and thus will not be addressed. This is because it is integrated into the data loading functionality, which is also not addressed in this iteration. Likewise, the DirectoryUtilities, which encapsulates some basic file system functionality, is only used within the persistence subsystem to support Model conversion and is therefore irrelevant at this point. See Section 5.3: *Loading, Converting and Dumping Persisted Data* on page 39 for a more complete description and analysis of Model Conversion.

## 6.7 Design – Database's Internal Structure

Lastly, the database internal collection structure must be defined. The eXist-db uses a hierarchy of collections and path definitions similar to the file system. The `/db` collection is the root collection and there is one predefined sub-collection, the `/db/system` collection. The `/db/system` collection maintains an XML resource defining the users and their passwords for the database and has one further sub-collection, `/db/system/config`, which provides configuration support for implicit validation and triggers. The .getmore persistence layer will be encapsulated in a sub-collection of the root addressed as `/db/getmore`. The .getmore collection is further broken into two sub-collections. The first sub-collection, `/db/getmore/data`, will store the Models and a second sub-collection, `/db/getmore/schemas`, will store the XSD schemas used for validation. Figure: 6.4 provides a visual depiction of the eXist-db's collection structure for supporting the persistence subsystem's needs.

- /db
  - /db/getmore
    - /db/getmore/data
    - /db/getmore/schemas
  - /db/system
    - /db/system/config

**Figure 6.4:** eXist-db Collection Structure

## 6.8 Design Result Overview – Persistence Subsystem

Excluding Model conversion, all dependencies on the java.io.File class may be realized through the previously defined StorageComponent and XSValidator interfaces. An updated class diagram of the persistence system with the new classes is depicted in Figure: 6.5. The StorageComponent class has replaced the file system, the Converter and Dumper subcomponents have been removed and the XSValidator class has replaced the XMLschemaValidation class. The FileAccessObject is no longer defined as an abstract class in order to facilitate testability.

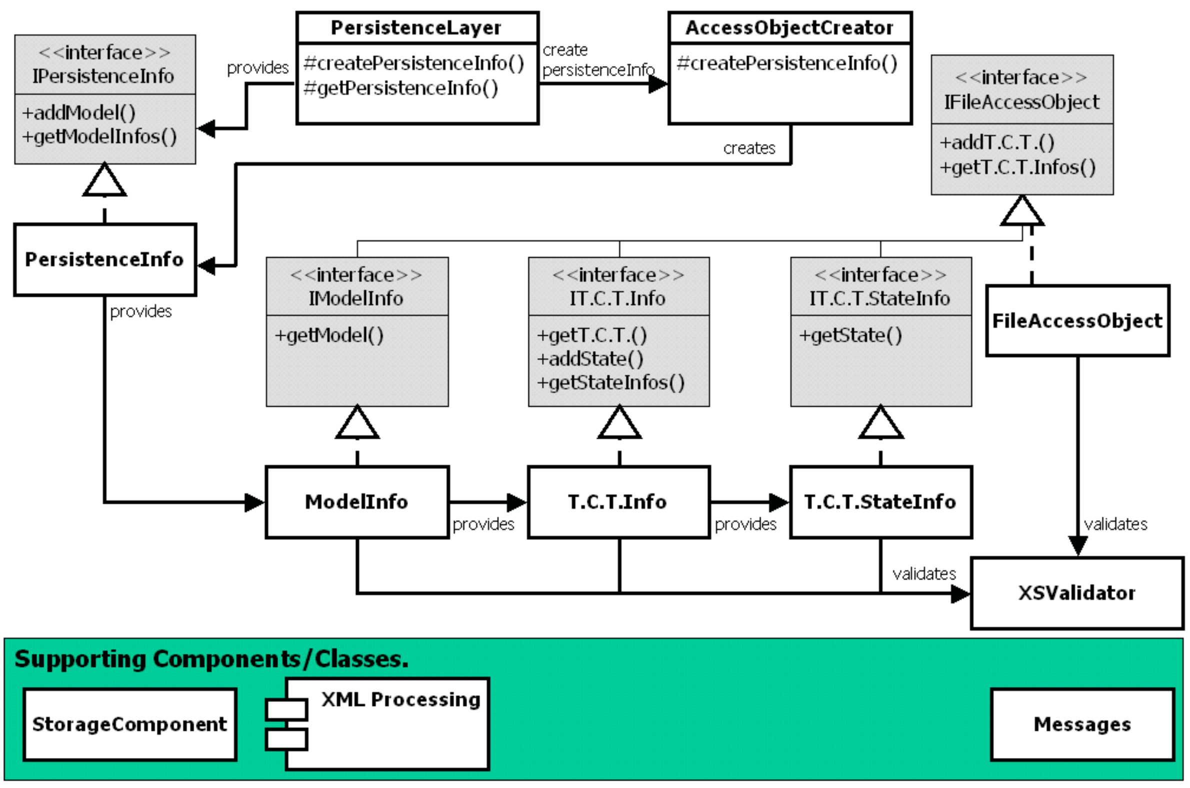


**Figure 6.5:** Persistence Subsystem's Class Diagram – Iteration One

## 6.9 Implementation – Proceeding and Selected Details

Implementation in the first iteration was accomplished in three phases. First, the StorageComponent was implemented. Next, new implementations of the PersistenceInfo, FileAccessObject, ModelInfo, TestCaseTreeInfo and TestCaseTreeStateInfo classes were developed and the AccessObjectCreator was updated to return the new PersistenceInfo implementation. Finally, the XSValidator was implemented and the system's primary classes were refactored to use the new validation functionality.

### Implemenation – Persistence Subdirectory

The implementation of the StorageComponent dealt first with accessing the eXist-db as an embedded database and then provided a façade over the XML:DB API. In order to use the eXist-db in embedded mode the appropriate Java Archives (JARs) need to be added to the classpath and working directory structure arranged. The twenty JARs that must be added are readily available in the basic database installation. Next a `persistence` directory was added to the .getmore system's structure and two key files, `conf.xml` and `log4j.xml`, were added to the target directory. The `conf.xml` file provides all configuration details to the database when it is started. It specifies, among other things, the directory where the data and journaling files will be maintained. They are both maintained typically in a subdirectory named `data` in the target directory, but that may be defined in the `conf.xml`. The name of the directory was changed to `storage` to avoid confusion with the file system's `data` persistence directory. Figure: 6.6 depicts the resulting directory structure.

- persistence/
  - data/ *– File system storage.*
  - storage/ *– Database storage.*
  - lib/ *– eXist-db JARs.*
  - conf.xml
  - log4j.xml

**Figure 6.6:** The Peristence Subdirectory Structure

### Implementation – Accessing the eXist-db

Next the code to startup and shutdown an instance of the embedded database was implemented. The general process of accessing the database through the XML:DB API is very similar to that defined in the JDBC API. The challenging portion was actually pointing the new database instance at the correct target directory, where the `conf.xml` file can be found. First the .getmore system's working directory must be obtained. This can be accomplished through a static call to the ConfigDirLocator class, which provides the path to the configuration directory, which is a sibling directory of the `persistence` (exist home) directory. Once the correct directory is ascertained, the database must be redirected. This can be accomplished through two means. Either through the use of an environmental variable or by setting a property on the database instance before it is registered. The setting of the database's property was assessed to be the most stable method and was used. When the database is registered with the DatabaseManager, it parses the `conf.xml` and `log4j.xml` files, conducts necessary initialization steps and is then available to be accessed.

After the database instance has been registered, a collection may be requested from DatabaseManager. The request is parameterized with the collection's Uniform Resource Identifier (URI), a user's name and password. The URI, commonly referred to as the address, for a collection is derived from the access method, database instance's name, host address, port and path to the desired collection. Figure: 6.7 depicts the activites for accessing the eXist-db database. A number of services, including the DatabaseInstanceManager, may be requested from a collection instance. The DatabaseInstanceManager provides the functionality to shutdown a database instance.

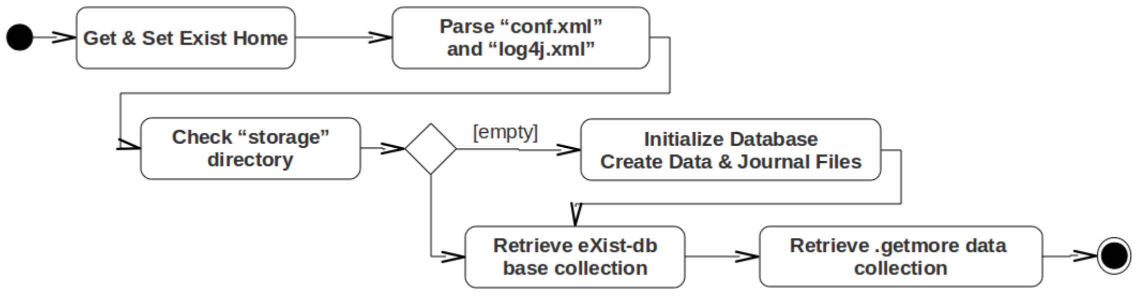

Get & Set Exist Home
Parse "conf.xml" and "log4j.xml"
Check "storage" directory
[empty]
Initialize Database Create Data & Journal Files
Retrieve eXist-db base collection
Retrieve .getmore data collection

**Figure 6.7:** eXist-db Access Activity Diagram

## Implementation – StorageComponent Façade

Once the basic database startup and shutdown functionality was implemented in the StorageComponent, the effort was turned to developing a façade over the XML:DB API. The Collection class of the XML:DB API provides the majority of functionality required. A sub-collection, parent collection or contained resource may be accessed. The path to the collection and a listing of all sub-collections and resources may be obtained. New resources may be created, stored and removed. An instance of the CollectionManagementService may be requested, which is capable of creating and removing collections. The resource, accessible from its parent collection, provides the necessary functionality to set and retrieve the resource's content. Resources may be either binary or XML resources. XML resources provide the ability to store DOMs and parse the content with a SAX content handler. Binary resources are used to store documents that are not well-formed XML

### Implementation – DAOs and XSValidator

Once the StorageComponent was complete, a new set of classes was developed to implement the persistence layer's interface and functionality. This was a rather straightforward process. The PersistenceInfo, FileAccessObject, ModelInfo, TestCaseTreeInfo and TestCase-TreeStateInfo classes were implemented. Next, the AccessObjectClass was refactored to return an instance of the new PersistenceInfo class.

The last phase of development consisted of implementing the XSValidator class. This is accomplished by first requesting an instance of the XQueryService from a collection. This service may then be used to execute an XQuery statement using the functionality provided by the JAXV extension functions. First, the validation was developed to only return whether the document was valid or not, but then was expanded to return also a complete validation report. This functionality was then built into the persistence layer's DAO classes.

### Implemenation – Installation Integration

At this point, the implementation of the first iteration was complete. The last step consisted of integrating the eXist-db into the installation process. The majority of the build and installation process is automatic. However, the directory structure needs to be defined in the configuration files and the correct JARs need to be added to the plugin's configuration file's runtime classpath definition.

## 6.10 Objectives Accomplished in Iteration One

The first iteration resulted in the replacement of the file system dependant persistence subsystem with one supported by an embedded eXist-db. The majority of the peristence layer's classes were reimplemented and two new classes, the StorageComponent and XSValidator, were introduced to to encapsulate the necessary eXist-db functionality. Figure: 6.8 depicts the resulting .getmore system. The overall structure remains the same, but the file system has been replaced by an embedded eXist-db.

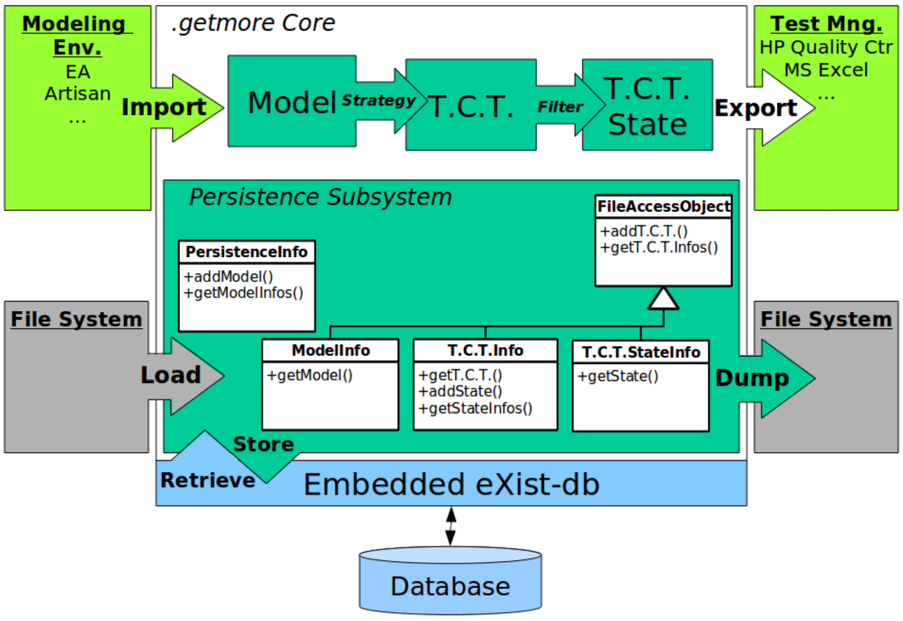


**Figure 6.8:** Resulting .getmore System – Iteration One

## 6.11 Summary

In this chapter the applied agile method and the first iteration's design and implementation was described. The first iteration dealt primarily with the replacement of the file system persistence method with a database implementation. The iteration's scope, overarching concerns and starting point were specified. Two design variants for the replacement of the java.io.File class were introduced and compared. It was decided that the eXist-db's functionality should be encapsulated in a separate StorageComponent class and used by the persistence subsystem's DAOs. Support for validation was provided by the XSValidator class. Next selected aspects of the design's implementation were covered. Finally the resulting .getmore system was presented.

# 7 Development – Iteration Two

This chapter will examine the developmental effort of the project's second iteration. The iteration's scope, design and implementation will be explained. Then an assessment of the developmental effort will summarize the results.

## 7.1 Scope of Iteration Two

The second iteration's focus was the accomplishment of the project's second goal: Provide local and remote access for multiple users to a common set of data. This goal is specified further through requirement five. Requirements 7, 8, 9, 10, 11, 15 and 16 were also identified for implementation in this iteration. See Section 3.1: *Requirements* on page 7 for a complete description of the requirements.

**R5:** Provide local and remote access for multiple users.

**R7:** Provide Model versioning control.

**R8:** Integrate the database configuration into .getmore's installation process.

**R9:** Convert Models created by other .getmore versions.

**R10:** Dump persisted Models to the file system.

**R11:** Load Models from the file system into the database persistence subsystem.

**R15:** Support Unique Identifiers (UIDs) for element identification.

**R16:** Support an extended internal data structure.

Requirements 5, 8, 10 and 11 were the base requirements expected to be completed. Requirements 7, 9, 15 and 16 were set as slack requirements to be implemented if sufficient time is available.

The end state of this iteration was defined as follows: Multiple users are able to access and manipulate a common set of persisted Models either locally or remotely. Users can dump Models from the persistence layer to a designated file system directory or load them from a specified directory into the persistence subsystem. Models are checked for correct versions prior to loading and during the initialization of the .getmore system. Any Models of an older version will be converted into the newer version. Models with an extended data structure can be persisted and elements are identified by UIDs provided by the .getmore Core. A user may choose to save a Model's state of development and later revert to this saved state. The database is integrated into the .getmore system's installation process to facilitate automatic product production and installation.

## 7.2 Design – Overview

The design effort in the second iteration was broken into four phases:

1. **Model Loading, Converting and and Dumping.** The first effort consisted of expanding the functionality of the database persistence method by including the Model conversion, loading and dumping capability. Due to the knowledge gained in the first iteration, this was determined to be the set of requirements (Requirements 9-11) with the lowest risk and the greatest payoff.
2. **Local and Remote Access.** The second effort consisted of providing local and remote access to the database (Requirement 5). This was the project's secondary goal and primary effort for the second iteration.
3. **Support Future .getmore Version.** The third effort consisted of supporting the .getmore system's next version. Specifically the support for UID identification and extended data structures. This consisted of Requirements 15 and 16.
4. **Extended Requirements.** The last effort was development of a version control system (Requirement 7) and laying the groundwork for a filtering mechanism. These requirements were lowly prioritized were thus relegated to the end of the work queue.

## 7.3 Design – Loading, Converting and Dumping Models

Loading refers to the transfer of data from the file system into the persistence system, dumping refers to the transport of data from the persistence subsystem to the file system and conversion refers to the transformation of a Model created by an older version of .getmore to a newer version's format. See Section 5.3: *Loading, Converting and Dumping Persisted Data* on page 39 for a more in-depth review of the legacy system.

### 7.3.1 Flexible Data Transport

First a general solution for data loading and dumping was designed. Intelligent derivation of intent was used to provide flexibility to the transfer of data between the file system and database. Data transport was envisioned to be similar to the flow of data through a pipe. However, the design was modeled after the pipe and not the flow of data from one location to another location, as is the normal perspective. The ends of the pipe are identified to be locations within the database and the file system and the data flows from one location to the other depending on the needs of the system. This basic conceptualization facilitates a more flexible alternative to data transport.

Depending on the specific targets set, an intelligent determination of the data flow may be determined. The target may be a file or directory in the file system and either a resource or collection within the database. Depending on the set of targets the appropriate functionality must be executed or identified as not possible. There are eight possible combinations of sources and targets, six of which are possible. It is not possible to dump a collection into a file or load a directory into a resource. However, all other combinations are possible. Table: 7.1 shows the possible combinations of targets. Each combination, the loading (top) and dumping (bottom), is identified as possible or not.

| | **Resource** | **Collection** |
|---|---|---|
| **File** | load – yes / dump – yes | load – yes / dump – no |
| **Directory** | load – no / dump – yes | load – yes / dump – yes |

**Table 7.1:** Data Transfer Target Combinations

Once the flow of data into and out of the database could be controlled, the functionality was extended to deal specifically with Models. First a collection must be tested to determine if it contains a properly formatted Model prior to dumping and likewise a directory must be tested to determine if it contains a correctly formatted Model prior to being loaded into the persistence subsystem. Additionally, in order to avoid naming conflicts as described in Section 5.3: *Loading, Converting and Dumping Persisted Data* on page 39, the loading of Models must support three different handling methods for dealing with naming conflicts. It must be able to ignore the new Model, overwrite the old Model or automatically create a sequentially incremented Model name.

The XSFSDataPort class provides the primary functionality for loading and dumping data. The XSFSModelLoader extends the data ports functionality in order to provide the Model specific loading functionality. The XSFSModelDumper does the same for dumping Models. It was determined that the loading and dumping of Models be separated in order to facilitate and delineate their usage for other developers. Figure: 7.1 depicts the class diagrams of the XSFSDataPort and the XSFSModelDumper and XSFSModelLoader classes.

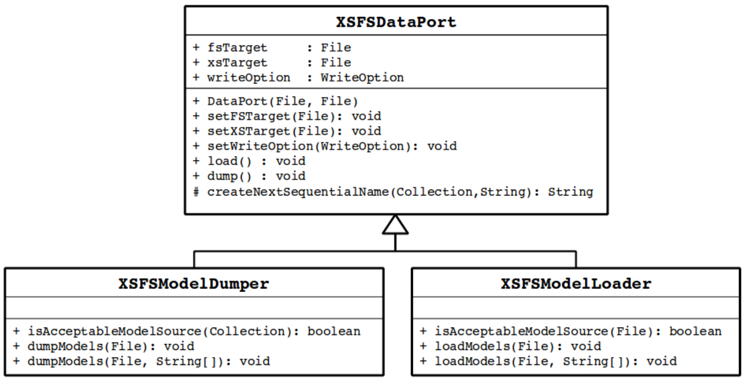

XSFSDataPort
+ fsTarget : File
+ xsTarget : File
+ writeOption : WriteOption
+ DataPort(File, File)
+ setFSTarget(File): void
+ setXSTarget(File): void
+ setWriteOption(WriteOption): void
+ load() : void
+ dump() : void
# createNextSequentialName(Collection,String): String
XSFSModelDumper
+ isAcceptableModelSource(Collection): boolean
+ dumpModels(File): void
+ dumpModels(File, String[]): void
XSFSModelLoader
+ isAcceptableModelSource(File): boolean
+ loadModels(File): void
+ loadModels(File, String[]): void

**Figure 7.1:** Model Loading and Dumping Class Diagram

### 7.3.2 Model Dumping

The .getmore system uses the DataExporterDialog class to interact with the user through a GUI and manage the dumping of Models. It requests an instance of the Exporter class through the ConverterMng class and gathers required information from the user. Next it requests that the Exporter instance dump the specified Models to the specified directory.

In order to support the database persistence subsystem, there are two methods that may be applied. The Exporter class may be extended to provide the new functionality or an interface could be defined which specifies the functionality and then a new implementation could be provided to support the database's dumping needs. The Programming-to-Interface approach was chosen as it provides a clearer separation of concerns, which will simplify the implementation and maintenance efforts.

Figure: 7.2 depicts the collaborating classes involved in dumping Models from the persistence subsystem to a designated directory. The highlighted classes are those that were updated from the base design. The most important change is that the DataExporterDialog accomplishes its tasks through the IModelDumper interface, which is implemented by the XSFSModelDumper and Exporter classes. This provides the necessary level of abstraction to allow a common framework to support both the file system and database persistence subsystems. Additionally, the ConverterMng must be able to determine which persistence method is currently being applied. This can be accomplished by checking the type of the PersistenceInfo currently being used by the PersistenceLayer.

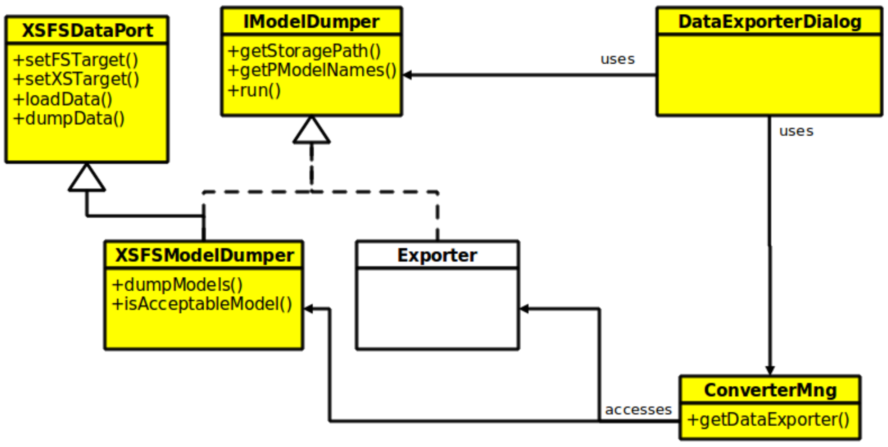


**Figure 7.2:** Model Dumping Class Diagram

### 7.3.3 Model Converting and Loading

The converting and loading subcomponents exhibit high coupling. In order to support Model conversion and loading for the database implementation, there were two primary options considered. Either the entire conversion and loading system may be replaced with a new implementation or the current implementation could be modified to be variably dependant on the current persistence method and the functionality provided by the XSFSModelLoader. The second approach was chosen due primarily to the disparity of involved effort estimations. Completely replacing the implementation was estimated to require two week's worth of effort, while modifying the current implementation was estimated to require a half-week's effort.

The general flow for Model conversion and loading starts with identifying the Models to be loaded and determining which Models must be converted. Next, the outdated Models must be converted and then the Models are loaded into the persistence subsystem. Lastly, the Model proxies must be refreshed to reflect the new state of the persisted Models.

When conversion is conducted as part of the startup the PersistenceInfo must parameterize the ConverterConfig with the directory where the models to be converted can be found. If the user chooses to load a Model from the file system, the DataConverterDialog is responsible for providing the configuration details. Once the details are gathered, the conversion is requested from the ConverterMng class. The flow of control must be conditionally controlled based on the type of persistence currently being used, which may be determined as described in the previous section. Simple calls to the java.io.File class are made to support the file system persistence method and the #loadModels() method is requested from the XSFSModelLoader to support the database persistence method. Finally, either the ConverterMng or the XSFSModelLoader notifies the folder proxies of the updates depending on the current persistence method. Figure: 7.3 on the next page depicts the classes that collaborate to convert and load Models into both the file system and database persistence subsystems.

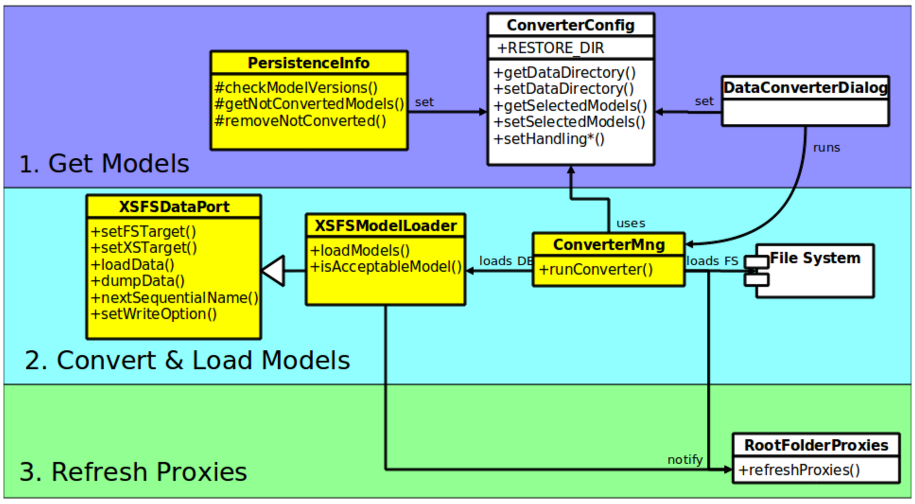


**Figure 7.3:** Model Conversion and Loading Class Diagram

## 7.4 Design – Local and Remote Access

Once the design to support the Model conversion, loading and dumping was complete, the focus was shifted to providing local and remote access to the data for multiple users. As described in Section 4.3: *Database Deployment Techniques* on page 22, in order to provide remote access to the persisted data, the database must be employed as a database server. Figure: 7.4 on the following page is an adjusted depiction of the deployment possibilities discussed in that section. This figure provides a application specific depiction of the employment possibilities of the eXist-db in support of the .getmore system. The left side depicts the eXist-db being deployed as an embedded database. The right side depicts the eXist-db being employed as a database server. A local .getmore client is one that is running on the same machine as the database server and a remote client runs on a remote host and connects to the database server over the Internet. In this diagram, the term “process” was used, instead of the term “Java Virtual Machine (JVM)” to generalize the separation. Since both the .getmore system and eXist-db are implemented in Java, they execute by a JVM.

The eXist-db may be deployed as a standalone server or in a Servlet container. The standalone server provides access networking interfaces described in Section 4.4.2: *eXist-db* on page 24. The database may also be deployed as part of a web application supported by a Servlet container such as Jetty or Apache’s Tomcat. The standalone

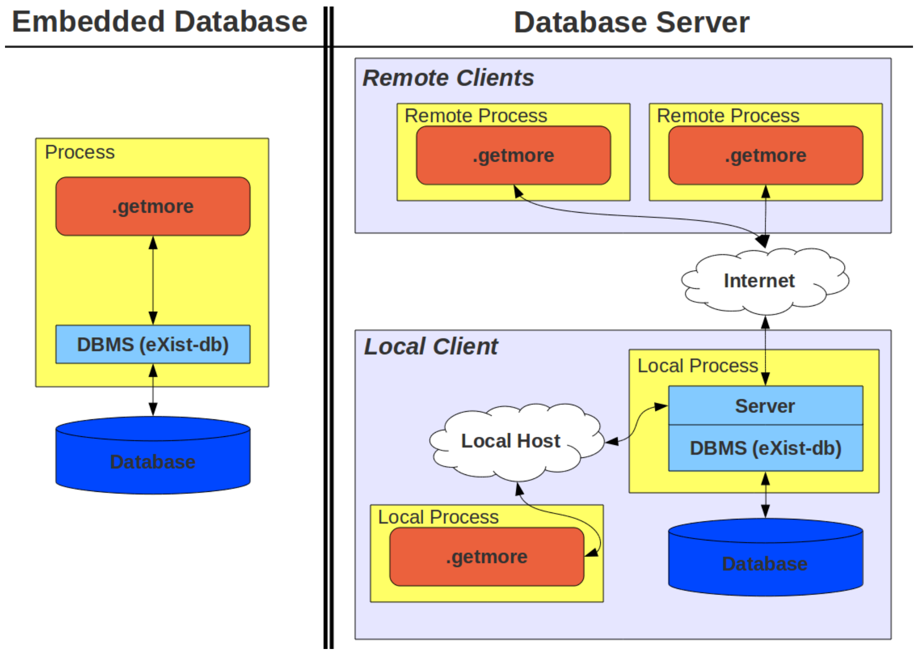


**Figure 7.4:** Database Persistence Methods Overview

server is based on a stripped down version of the Jetty web server and is configured through a `server.xml` file found in the server's working directory. It is more reliable and provides a performance gain over web application setup. However, it lacks the additional services available in the Servlet environment. The persistence needs of the .getmore system are not such that a complete web application would be appropriate; therefore the database will be deployed as a standalone server. Accessing the database as a server increases the complexity of the StorageComponent access methods, but does nothing to change the actual overall design.

### 7.4.1 Dynamic Persistence Framework

While reviewing the requirements, it was realized that the various persistence methods could be integrated into an overall persistence framework. To allow maximum flexibility and support, the currently employed persistence method should be dynamically determined based on the user's desires and the availability of the supporting persistence methods. The variable modes of the database's deployment should be controlled through a set of user-defined properties; additionally this set could then be extended to also include support for file system persistence. Through combining all these various methods the persistence subsystem would present a flexible, fault-tolerance persistence framework which could be managed by the .getmore system as needed.

The resulting four methods of persistence are:

**File System.** Persists data as files in the file system using the java.io.File class. Only provides local single client access to the persisted data. Provides a least common denominator for persistence.

**Embedded Database.** Persists data as resources within an eXist-db. The eXist-db is embedded within and shares a common JVM with the .getmore system. Only provides local single client access to persisted data.

**Local Database Server.** Persists data as resources within an eXist-db. The database server is started locally on the client's machine and provides access to multiple local and remote clients simultaneously.

**Remote Database Server.** Persists data as resources of a remote eXist-db. Provides access to, but not control over, a remotely started eXist-db server.

### 7.4.2 Persistence Properties

A set of properties identifying the persistence method will be stored in a file located in the persistence subdirectory. The set of persistence properties will also provide the address for the desired database server, the user's name and the user's password in order to support local and remote access to a database server.

The PersistenceProperties class, depicted in Figure: 7.5 on the next page, ties the designated property file to an interface for accessing and manipulating the properties stored within the file. The class also provides the functionality to load/reload the properties from the file and store the current set of properties to the file. An instance of this class is statically available to the entire .getmore system through the PersistenceLayer, which also provides static point of access for an instance of the PersistenceInfo class. This will allow either the .getmore system to dynamically alter the properties during runtime or the user to manually manipulate and configure the persistence layer.

In order to change the method of persistence, the system must first set the PersistenceProperties `persistence.method` and then request the creation of a new PersistenceInfo

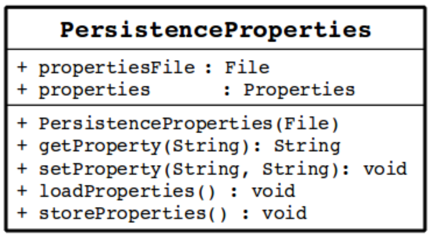

PersistenceProperties
+ propertiesFile : File
+ properties : Properties
+ PersistenceProperties(File)
+ getProperty(String): String
+ setProperty(String, String): void
+ loadProperties() : void
+ storeProperties() : void

**Figure 7.5:** PersistenceProperties Class Diagram

object through the PersistenceLayer. The creational process will be covered in more depth in Section 7.4: *Design – Local and Remote Access* on page 65.

## 7.4.3 Fault-Tolerant Persistence

Fault tolerance may be accomplished through redundancy of three aspects: functionality, time and information. Functional redundancy provides fault tolerance through use of diverse software applications, assuming if a software component fails a secondary diverse component may successfully accomplish the task. Temporal redundancy is accomplished through the re-execution of commands. The re-execution may be based on the same set of commands, assuming the failure was externally caused, or by an alternate branch, if the same error is expected to be encountered again during the re-execution. Informational redundancy provides fault tolerance by maintaining multiple copies of the information and comparing the results attained from operating on each set. It also provides protection from data corruption.

The fault tolerance in this persistence framework is primarily expressed through its functional redundancy, its multiple persistence methods. If a persistence method is not available, an alternate method should be used. Temporal redundancy is provided by the eXist-db's journalling and automatic recovery capability as well as the exception handling implemented in the associated code. Informational redundancy may have been implemented by synchronizing the "workspaces" or sets of data across all methods. Synchronization between the file system and database is a complex subject with many issues to be dealt with. Due to time constraints and priority of effort, informational redundancy will not be further considered or implemented.

Functional redundancy within the framework is employed through a fallback technique. If a local database server instance is not manually started before the .getmore client is started, then the client will request the start of a database server. The StorageServer encapsulates the database server functionality and is responsible for receiving the request and starting a local instance of the standalone database server. If no database instance is accessible, then the system will fallback to file system persistence. The file system persistence represents the simplest persistence method which supports the needs of the .getmore system.

Figure: 7.6 depicts the checks that are undertaken by the AccessObjectCreator before a PersistenceInfo is created. This flexible fallback strategy provides fault-tolerant persistence architecture for the .getmore system. This stategy will be further extended by allowing the all of the database persistence methods to operate on a communal set of data. How this is accomplished will be discussed in the next section.

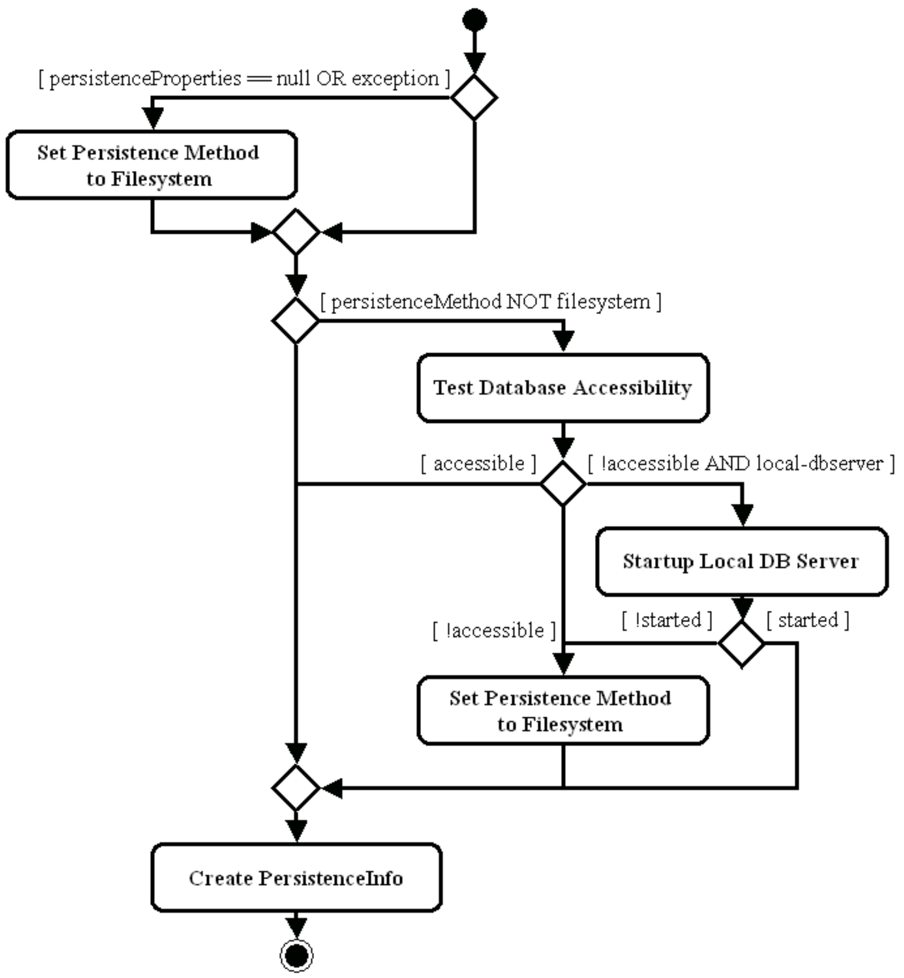

[ persistenceProperties == null OR exception ]
Set Persistence Method to Filesystem
[ persistenceMethod NOT filesystem ]
Test Database Accessibility
[ accessible ]
[ !accessible AND local-dbserver ]
Startup Local DB Server
[ !accessible ]
[ !started ]
[ started ]
Set Persistence Method to Filesystem
Create PersistenceInfo

**Figure 7.6:** PesistenceInfo Creational Activity Diagram

### 7.4.4 Persistence Directory

Next the layout of the persistence directory and configuration files was established. Within the `persistence` directory, the `data` subdirectory is provided for the file system persistence implementation. The `eXist-db` subdirectory contains the structure for the database persistence subsystem. The `temp` subdirectory provides a location for manipulating temporary files in the file system. The `persistence.properties` file designates the desired persistence method, the database's address, user's name and the user's password.

The most critical decision was how to deal with multiple persistence configurations within the `eXist-db` subdirectory. Originally, it was planned that the persistence methods would be separated into subdirectories and each would maintain a separate set of configuration files and manipulated data. This initial approach of separating the methods would be sufficient to fulfill the project's requirements, but it would be beneficial if the persisted data could be managed as a communal set instead of each method maintaining a separate set of data. This, combined with the use of the persistence properties, would allow a user to switch persistence methods depending on their needs and still manipulate the same set of data. A user may work independently through the embedded database, which provides a performance boost when accessing and manipulating the data, in order to import and develop a base set required of tests. Then the user may open the access as a local database server and allow remote users to access and continue to manipulate the created set of tests. This flexibility is not set forth by any requirement, but may be accomplished through an innovative integration of the deployment possibilities. This is realizable by providing multiple configuration files and specifying the name of the configuration file to be used when the database is created. Additionally, the data and journal files for each database instance are of the same format. Therefore, it is possible to employ the database in various modes, manipulating a single set of data. It was also determined that the library files necessary for the standalone server's deployment must be a subdirectory of the server's working directory. It is not sufficient to add them to the .getmore system's classpath, as the server runs in a separate JVM.

Within the `eXist-db` directory, the `lib` subdirectory contains all of the JAR files necessary for the eXist-db's usage. The `scripts` subdirectory provides scripts to startup and shutdown the standalone database server. The `storage` subdirectory contains the database's data and journal files. The `conf.xml` file provides the configuration for the database. The `sever.xml` file provides the configuration details for the standalone database server. Figure: 7.7 is a visual depiction of the persistence directory's layout.

- persistence/
  - data/ *– File system storage.*
  - eXist-db/ *– Database subdirectory.*
    - storage/ *– Database storage.*
    - lib/ *– eXist-db JARs.*
    - scripts/ *– Startup/shutdown database server.*
    - conf.xml
    - log4j.xml
  - temp/ *– Temporary files.*
  - persistence.properties

**Figure 7.7:** The Peristence Subdirectory Structure

## 7.5 Design – Separation of User and System Concerns

The separation of user and system concerns of the persistence subsystem must also be addressed when dealing with multiple users. The user specific concerns dealing with the display of the elements in the Model explorer need to be handled for each user separately. If all users were sharing the same expanded variable for a Model, it would continuously be expanded and collapsed as different users had different viewing needs. In order to deal with this problem, two approaches were considered. Either the user preferences could be managed in a local file external from the persisted Models or they could be migrated to an eclipse preference setting. The second approach was chose because it fit the model employed by the .getmore system and provided the best strategy overall support for managing the comprehensive set of user preferences. The effort to migrate the user concerns was determined to be a effort best implemented by the sepp.med GmbH and thus not further discussed in this work.

## 7.6 Design – Support for Future .getmore Version

Next the design effort focused on supporting the next version of the .getmore system. The next version of the .getmore system will use UIDs instead of the element's name as the primary method of identification and uses an extended internal data structure to allow Test Case Trees to contain both Test Case Tree States and other Test Case Trees and Test Case Tree States to contain both other Test Case Tree States and Test Case Trees.

### 7.6.1 UID Support

In the new version of .getmore, UIDs will be generated by the .getmore Core when the elements are created and remain immutable, whereas the display names used in the Model explorer were expected to change. The element's UID and name will both be stored in the element's `.metadata` file. The use of the instable names for identification led to difficulty within the addressing mechanism, as the element's addresses would change when the name of an ancestor element changed. In order to support using UIDs for element identification, two approaches were considered. The first method would leave the design of the persistence subsystem unchanged and simply change the purpose of the respective methods. All functions that were used to get and set the name would be used to get and set the UIDs. The second approach would refactor all naming methods to identification methods and new methods for altering the display name would be added to the interface. In both approaches, the methods that would be used to change the UID would be deprecated since the UIDs should be immutable. The second method was chosen because allows for consistency in the .getmore system's expectations, which is important in extending a legacy system.

Figure: 7.8 on the facing page depicts the refactoring of the IFileAccessObject and IModelInfo interfaces. The asterisks denote methods that are deprecated. Since names must no longer guaranteed to be unique, there is no corresponding method to determine if a name is used.

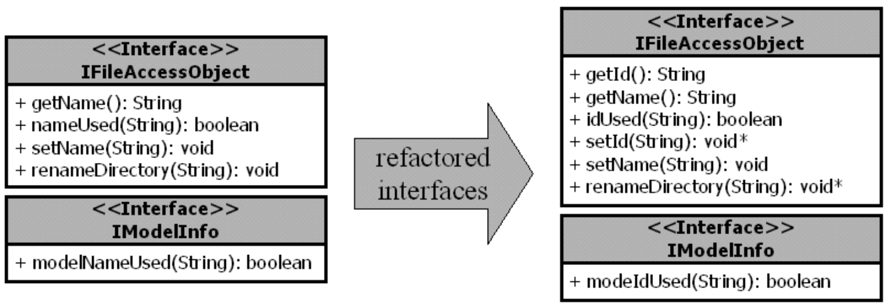


**Figure 7.8:** Refactored Interfactes to Support UID Identification

## 7.6.2 Extended Data Structure Support

In order to support the extended internal data structure of .getmore's next version the persistence layer's interface must be extended. An unexpected benefit from the initially questionable interface structure was discovered. Since the methods to add Test Case Trees and get the set of Test Case Tree DAOs is defined in the IFileAccessObject, they are inherited and available both in the ITestCaseTreeInfo and ITestCaseTreeStateInfo interfaces. However, support for adding a Test Case Tree State and getting the set of Test Case Tree State DAOs must be added to the ITestCaseTreeStateInfo interface. This is done through adding the two respective methods to the interface. The IFileAccessObject interface is not extended, because the Model DAO should not be able to add or retrieve Test Case Tree State DAOs.

Figure: 7.9 depicts the refactored ITestCaseTreeStateInfo interface with the two new methods for including a new Test Case Tree State as a subelement of a Test Case Tree State.

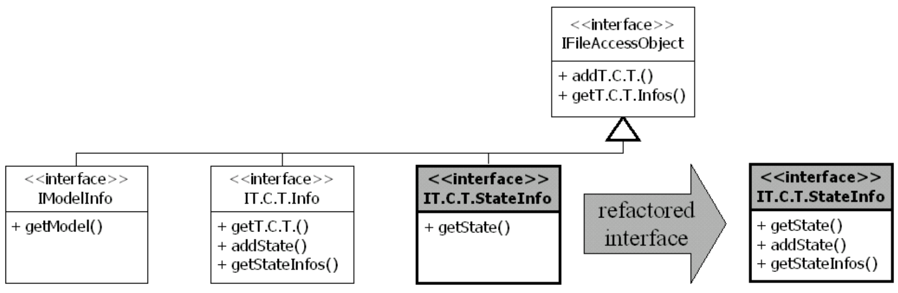


**Figure 7.9:** Extended ITestCaseTreeStateInfo Interface

## 7.7 Design – Model Version Control

After the completion of the design supporting the use of UIDs and the extended data structures, the focus was turned towards providing versioning control for the Models. Requirement 7 defines the expectations of Model versioning control. This functionality is not currently used by the .getmore system, but the functionality will be included into the persistence subsystem to support future versions of the .getmore system. There are several concerns that must be supported by a versioning system. First the state of a Model must be maintained so that it may be restored at a later point. Second a list of stored states or revisions must be maintained, so that the system can identify which previous states are available. Finally the system should be able to determine if a Model has been changed to determine if the current Model's state represents a new revision.

### 7.7.1 Versioning Design Decisions

There are multiple implementations of version control available through open source projects such as Subversion (SVN) or the Concurrent Versions System (CVS). However, these implementations are designed to work with the files in a file system and are thus not appropriate for the versioning of data stored in an XML database [PCSF08]. Additionally, eXist-db provides a versioning module for versioning resources within the database. This extension provides the ability to track changes to a document and store the differential between revisions. However, this implementation is only appropriate for files not larger than a couple of Megabytes and does not recognize changes made through XQuery update extensions or XUpdate [eXi10][1]. Because of these limitations, the eXist-db versioning module was also deemed inadequate for the needs of the Model versioning system. Therefore, it was determined that a new prototypical versioning system be designed to support the future needs of the .getmore system. An advanced versioning system represents an entire project within its own rights well beyond the allocated effort within this project, but the basic versioning functionality is within reason.

1 eXist-db Versioning: `www.exist-db.org/versioning.html`

#### Version Granuality and Storage

First the granularity of the versioning system was considered. Depending on the level of granularity employed by the system, changes may be tracked at the Model level or at the subelement level. Tracking changes at the Model level is simpler but the tracking at the subelement level would provide an increased flexibility. It would be possible to revert just a desired Test Case Tree or Test Case Tree State instead of the entire Model. A finer granularity of versioning is a valuable asset, but it was deemed to require more than the effort allocated to this requirement. Therefore, the changes and revisions would be maintained at the Model level in this design.

The storage of a Model's current state may be accomplished simply by maintaining complete copies of the Model revisions or may be accomplished through a complex differential algorithm, which stores only the changes to the Model since the last committed revision. The first approach is simple but requires significantly more storage space. The second approach requires less space but the restoration of a previous version requires a significantly more involved process to recreate the desired state. Because of available effort limitations, the first method was chosen.

Due to the previous two decisions, it could occur that the maintained Model revisions may require extremely large amounts of space. Therefore, a mechanism to control the number of revisions stored by the system must be employed. This mechanism would set the total number of revisions that would be maintained by the system and as new revisions were committed, the oldest revisions would be removed from the versioning system. This is a functionality not typically associated with a versioning system and should be controllable by the system. The system should be able to dynamically set the number of revisions maintained based on the user's expectations and available resources.

### 7.7.2 Versioning Design

A Model may be in one of three different states: *Unversioned*, *Unchanged* or *Changed*. First an *unversioned* Model must be added to the versioning system. Adding the Model creates a first revision and registers the trigger for the collection. It is then considered in the *unchanged* state. When the Model is then changed the Model's status must be updated to reflect the new *changed* state. Later, the *changed* Model may be committed or reverted. If the *changed* Model is committed its current state is stored and resets the

Model's status to *unchanged.* If the Model is reverted, the system must first determine if the requested revision is available. If the revision is available, the Model is reverted to the designated state and its status is reset to *unchanged.* A Model may also be removed from the versioning system, any future changes are ignored and it reenters the *unversioned* state. Figure: 7.10 depicts the state diagram of the versioning system.

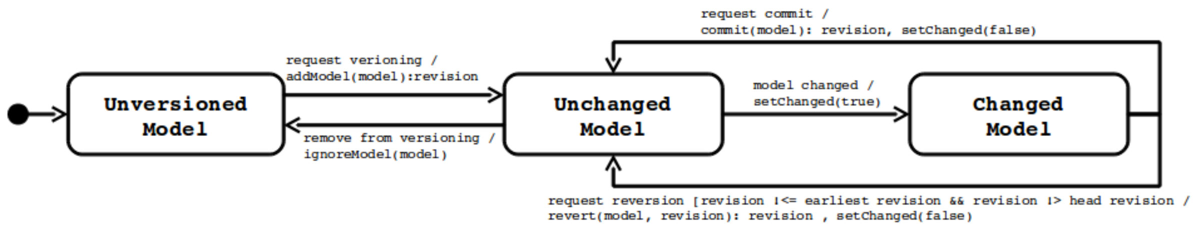


**Figure 7.10:** Versioning State Diagram

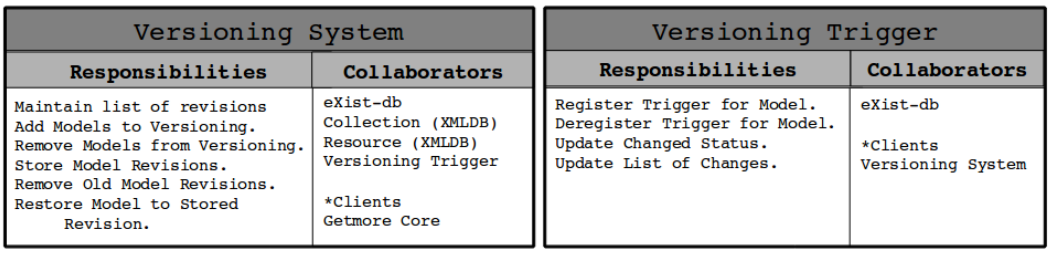
## Tracking System Changes

Next the mechanism to track changes to a Model was considered. The best mechanism to track changes within the eXist-db is the trigger. Triggers may be designed to respond to either changes of a resource or collection. There are a total of six defined events that correspond to the manipulation of a resource or collection. The three resource events are: store, update and remove. The three collection events are: create, rename and delete. An eXist-db trigger is fired twice, once before an event is enacted and once after the event has been completed. In order to use a trigger in the eXist-db, the trigger must first be registered. This is accomplished by creating a collection specific configuration resource in the database's configuration collection. The configuration collection mirrors the structure of the database's collections and a configuration resource defining the trigger is stored in the respective collection. All descendants in the configuration collection's hierarchy for which it was registered inherit the policy defined by the configuration resource. Any descendant may redefine the policy through its own configuration resource. The resource is named `config.xconf` and is an XMLResource defining the list of recognized triggers and the list of events that will cause the execution of the trigger and the location of the response to be executed by the trigger. The eXist-db supports both Java and XQuery triggers, which respond to events occurring within the database. Implementing the CollectionTrigger or DocumentTrigger interface or extending one of the triggers implemented as part of the eXist-db may define a Java trigger class. In order to define an XQuery trigger, a corresponding XQuery resource must be defined which executes the desired functionality [eXi10][1].

First the eXist-db's available trigger implementations were considered. The HistoryTrigger automatically stores old versions of a document in a history collection before it is overwritten or removed. However, the versioning system should only store the state of a Model when it is specifically requested by the .getmore system. The HistoryTrigger would waste system resources through unnecessary copying of resources. Additionally, it only implements the DocumentTrigger and is not designed to recognize changes to a collection. The other implementations were also deemed unsuitable. Therefore, it was decided that a new trigger should be designed. The trigger is responsible for updating the system when a Model changes. It will update the Model's status and a list of changes to the Model to support expected development in the future.

## Versioning and Trigger CRC Cards

The system is responsible for basic versioning functionality and the trigger is responsible for updating the system of changes. Figure: 7.11 depicts the distribution of the responsibilities between the versioning system and trigger. Respective classes were then derived from the CRC Cards.



| Versioning System | |
|---|---|
| **Responsibilities** | **Collaborators** |
| Maintain list of revisions<br>Add Models to Versioning.<br>Remove Models from Versioning.<br>Store Model Revisions.<br>Remove Old Model Revisions.<br>Restore Model to Stored Revision. | eXist-db<br>Collection (XMLDB)<br>Resource (XMLDB)<br>Versioning Trigger<br><br>*Clients<br>Getmore Core |

| Versioning Trigger | |
|---|---|
| **Responsibilities** | **Collaborators** |
| Register Trigger for Model.<br>Deregister Trigger for Model.<br>Update Changed Status.<br>Update List of Changes. | eXist-db<br><br>*Clients<br>Versioning System |

**Figure 7.11:** Versioning System and Trigger CRC Cards

1 eXist-db Triggers: `www.exist-db.org/triggers.html`

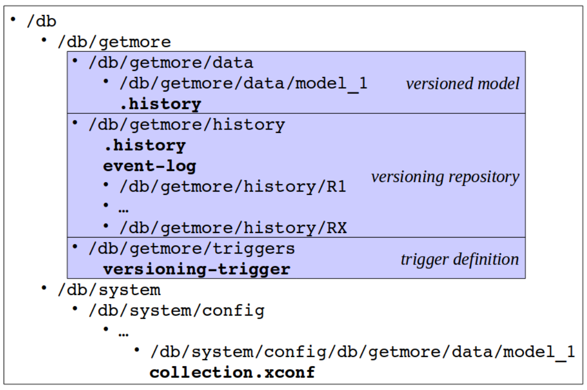
### 7.7.3 Versioning Layout

Finally, the actual layout of the versioning system was considered. The layout is similar to that used by most version control systems. There will be a base repository responsible for maintaining all revisions and the system's information such as the changes that were made and the current head revision. A single revision count will be maintained at the system level and Models may or may not have be involved a specific revision. Therefore, each Model will also maintain a history indicating which revisions it was part of. This will reduce the effort in determining if a Model may be reverted to a specified revision. The maintained history will also indicate whether the Model has been changed since the last commit or revert and the current revision of the Model since it does not necessarily have to be the most recent revision. The actual trigger implementation will be maintained in a separate trigger collection of the database and the configuration files will be registered in the appropriate Model directory as previously discussed.

Figure: 7.12 on the facing page depicts the described versioning subsystem's layout. The layout is broken into three parts. The first is the Model that is under version control. It contains a `.history` file storing Model specific versioning details. The versioning repository contains a `.history` file of all executed commits, an `event-log` detailing the events that were triggered in the system and a sub-collection for each of the committed revisions. Each revision sub-collection, prefixed with "`R`" and suffixed with the global revision count contains a copy of the committed Model's states. The trigger to be executed is located in the `triggers` sub-collection of the `getmore` collection. It is located in a separate collection to facilitate the definition of other triggers supporting the .getmore database persistence method. Finally, there is a `collection.xconf` resource located in the versioned Model's respective configuration collection. The configuration file could be located in the parent "getmore" configuration collection. However, this will require additional filtering to be done by the trigger.

### 7.7.4 Versioning Actions

After a Model has been added to the versioning system it may be committed, reverted to a previous version, the current status of the Model can be requested or it may be removed from the versioning system. The system intentionally does not ensure that a Model has



```
• /db
    • /db/getmore
        • /db/getmore/data
            • /db/getmore/data/model_1
              .history
        • /db/getmore/history
            .history
            event-log
            • /db/getmore/history/R1
            • …
            • /db/getmore/history/RX
        • /db/getmore/triggers
            versioning-trigger
    • /db/system
        • /db/system/config
            • …
                • /db/system/config/db/getmore/data/model_1
                  collection.xconf
```

**Figure 7.12:** Versioning Subsystem Layout

been changed before it is committed. The system is designed with the facilities to allow the enacting client to enforce or not enforce the policy. This provides maximum control to the using client and assumptions about the usage are kept to a minimum. The triggering mechanism is depicted as a concurrent operation, which functions independently of the versioning system. Figure: 7.13 depicts the activities associated with the versioning system.

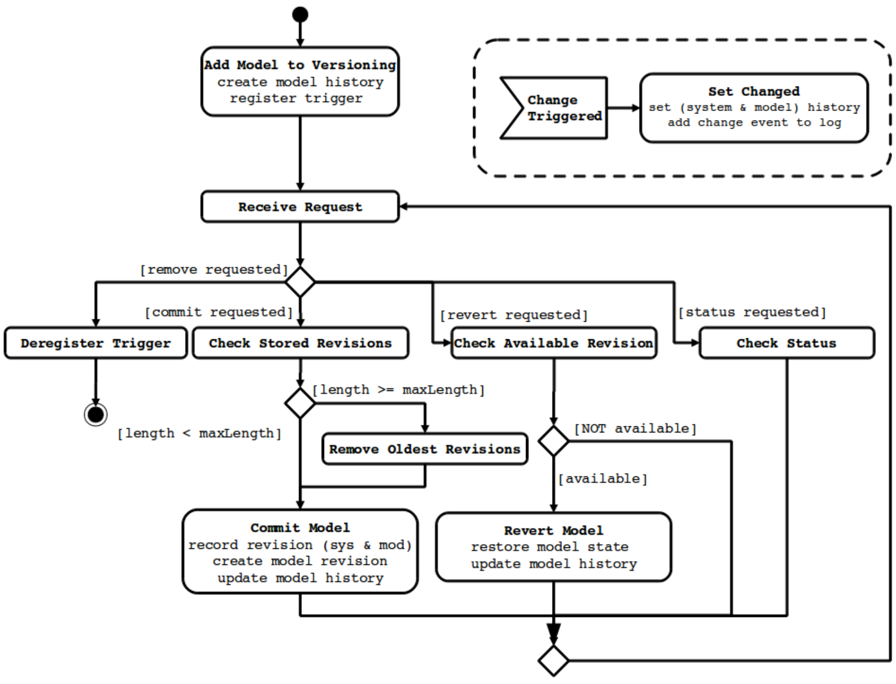


**Figure 7.13:** Model Versioning Activity Diagram

## 7.8 Design – XQuery Template

A last goal of the project was to lay the groundwork for a later filtering mechanism. XQuery is the technology perceived to most-likely support this development. eXist-db also supports a set of update extensions that may be used with the basic XQuery functionality. The update extensions allow for the deletion of a node, insertion of a node, replacement of a node, changing of an element's name and changing an element's value [eXi10][1]. The Design Pattern Template Method [GHJV95] could be applied to provide basic querying. To achieve this the XQuery can be envisioned as a basic framework of statements, each with a blank to be filled into to provide the specific querying details. A user should be able to define the resource, the target node within the resource, a filtering condition of the targeted node and an optional replacement node or value based on the type of query being executed. The XQuery Template will be able to return a set of nodes, return a count of nodes found, delete a node, insert a node, replace a node, rename an element or change the value of an element.

Listing: 7.1 depicts the XQuery template structured into an XQuery statement. The `RESOURCE` is the XML resource to be queried. The `TARGET-NODE` and `CONDITION` are both optional and if they are not present will be ignored, but are the primary points for querying support. The `UPDATE-EXTENSION` is derived from the given query type and the replacement expression.

```
for $target in doc( RESOURCE ){ TARGET-NODE }
{ where CONDITION }
return
    [ $target | UPDATE-EXTENSION ]
```

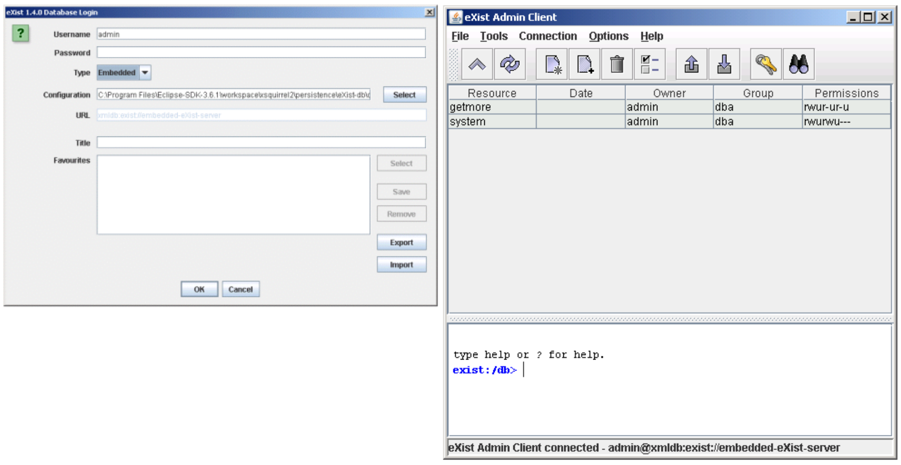
**Listing 7.1:** XQuery Template Format

1 eXist-db XQuery Update Extensions: `www.exist-db.org/update_ext.html`

## 7.9 Design – Administrative Client

Requirement 13 is to provide backup and restoration of persisted data and Requirement 14 is to provide an administrative client for the database. Originally, these were estimated to require significant effort. However, the eXist-db provides a Java implementation of an administrative client with an intuitive GUI. This administrative client may be used with an embedded database or remote database server. A user may use the client to access the database, alter user accounts, configure the database and directly manipulate the data stored within the database. Additionally, the client provides a mechanism for backing up and restoring data from a backup. Providing access to this administrative client fulfills Requirements 13 and 14.

Figure: 7.14 shows screen shots of the administrative client. The left screen shot is a shot of the log on frame. It provides a setting for accessing a database as an embedded database or remotely. The right screen shot is the client after it has accessed a database.



| | |
|---|---|
| Username | admin |
| Password | |
| Type | Embedded |
| Configuration | C:\Program Files\Eclipse-SDK-3.6.1\workspace\xsquirrel2\persistence\eXist-db\ |
| URL | xmldb:exist://embedded-eXist-server |
| Title | |
| Favourites | |

| Resource | Date | Owner | Group | Permissions |
|---|---|---|---|---|
| getmore | | admin | dba | rwur-ur-u |
| system | | admin | dba | rwurwu--- |

```
type help or ? for help.
exist:/db>
```

eXist Admin Client connected - admin@xmldb:exist://embedded-eXist-server

**Figure 7.14:** eXist-db Java Administrative Client

## 7.10 Design Result Overview – Persistence Subsystem

Figure: 7.15 on the following page depicts the resulting class diagram of the persistence layer after the second iteration. It combines the file system and database persistence implementations. The most important conceptual change to the base diagram is the addition of the PersistenceProperties class, which is used to control the subsystem's current persistence method. The DataPort component, which includes the XSFSModelLoader and XSFSModelDumper, was identified as a supporting component. Additionally, the StorageComponent was altered from a class to component to represent its growing set of responsibilities. It represents the StorageComponent and StorageServer classes and the supporting eXist-db.

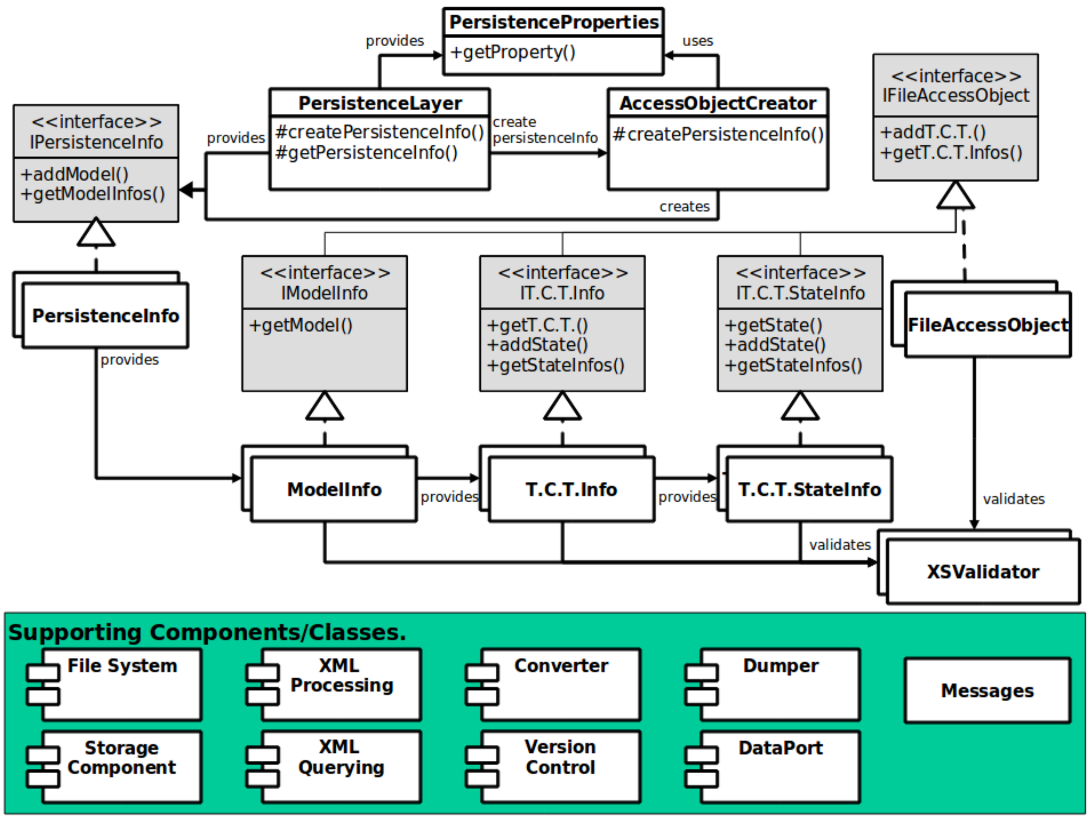


**Figure 7.15:** Persistence Subsystem's Class Diagram – Iteration Two

## 7.11 Implementation – Proceeding and Selected Details

Implementation in the second iteration was accomplished in phases as described in the iteration's design. The first phases was supporting Model conversion, loading and dumping. The second phase provides local and remote access to the persisted data. The third phase supports the planned .getmore system extensions. The fourth phase implements the Model versioning system, provides the groundwork for a future filtering mechanism and provides access to the administrative client.

## Implementation – Persistence Properties

Before any other development could be started PersistenceProperties class and its access through the PersistenceLayer was implemented. This provided a method for the persistence subsystem to dynamically determine the current persistence method to use. This was accomplished by first creating the `persistence.properties` file in the `persistence` directory and then implementing the PersistenceProperties class, which provided the functionality to load the properties from the file, query and set their current values and store the set of properties back to the file. Next a static method was implemented in the PersistenceLayer that instantiated the PersistenceProperties with the properties file and then returned the instance.

## Implementation – Model Conversion, Loading and Dumping

Next the XSFSDataPort was implemented. Special attention was paid to its flexibility, as it would be extended to support both the loading and dumping of Models. Once the data port was implemented the database was capable of loading and dumping general data. The general capability was then refined to support the dumping of Models from the database. In order to support the overall systems ability to dump Models either from the database or file system persistence subsystems, the IModelDumper interface was defined. The Exporter class was then refactored to implement the interface and the XSFSModelDumper extended the XSFSDataPort and also implemented the interface. The DataExporterDialog was reoriented to use the IModelDumper interface and the ConverterMng class was refactored to return the appropriate implementation based on the current persistence method defined in the PersistenceProperties.

Support for Model conversion and loading was implemented next. However, due to the coupling between the conversion, loading and file system, no clear separation was achieved. This would require a complete redesign of the system and was not part of this project. First the XSFSModelLoader class, which extends the XSFSDataPort class, was implemented. It provides Model specific support in determining if a designated directory is a correctly formatted Model and loading a Model into the database. Then the database implementation of the PersistenceInfo was refactored to support the checking and conversion of the Models persisted in the system when it was started. As described in the design, the Models are first exported to a temporary directory and then the versions are checked. Any Models that do not require conversion are then removed and the converter configuration is updated. This implementation follows the same sequence of steps taken by the PersistenceInfo class implemented for file system persistence. Next the #runConverter() method of the ConverterMng class was refactored. Depending on the current persistence method, different directories are used and after the conversion, the Models are loaded into the proper persistence subsystem.

### Implementation – Integrating Persistence Methods

Local and remote access to the persisted Models through employing the eXist-db as a database server was next implemented. First the directory was reorganized as defined in the design. The libraries were re-added to the classpath and plug-in configuration. The necessary scripts for starting and stopping the database server and accessing the administrative client were created. Next the StorageComponent's #startup() and #buildAddress() methods were extended to handle accessing the eXist-db in the required deployment methods. The StorageServer class was then developed to encapsulate the standalone server's functionality. The StorageComponent's #startup() method was extended again to start a new instance of the standalone server, through the StorageServer class, if the persistence method is local database server and the server is not accessible.

Finally, the fault-tolerance fallback technique described in the design was implemented in the AccessObjectCreator class. At this point the .getmore persistence framework supports all four methods of persistence. The internal data structures could be persisted in the file system, in an embedded database, in a local database server and in a remote database server. Additionally, the embedded database and local database server were integrated to manipulate the same set of data, providing user transparency between the methods.

### Implementation – UID and Extended Structure Support

In order to support UIDs for element identification, the IFileAccessObject and IModelInfo interfaces were refactored as described in the design and then the FileAccessObject and ModelInfo classes in both the database and file system persistence implementations were refactored to fulfill the interfaces contractual agreements.

Next the ITestCaseTreeStateInfo interface was extended, as described in the design section, in order to support the planned extended internal data structure of the .getmore system.

Once again the TestCaseTreeStateInfo class in both implementations were refactored to support the extended interface.

### Implementation – Version Control Subsystem

In order to implement the Model version control system, the Versioning class was first implemented. This class provided the base functionality needed for the versioning system. Next, the versioning system's database structure was created. The history and triggers sub-collections were created. Then the XQuery trigger was developed and loaded into the database's trigger collection. Finally the VersioningTrigger class was created in order to provided trigger registering and deregistering functionality.

### Implementation – XQuery Template

Lastly the XQueryTemplate was implemented to provide the groundwork for a filtering mechanism. The database persistence subsystem's FileAccessObject was then refactored to uses this functionality to set and retrieve an element's display name. The use of the XQueryTemplate provides better performance than the serial parsing, recreating and writing of the metadata file as is done in the file system persistence subsystem. The performance evaluation will be further described in Section 8.2: *Performance Evaluation* on page 90.

## 7.12 Objectives Accomplished in Iteration Two

The second iteration resulted in the introduction of a database-backed persistence method that provided local and remote access to multiple users. Additionally, it integrated the file system and embedded database persistence methods into an overall fault tolerant persistence framework. It also provides:

1. Model conversion, loading and dumping functionality for each persistence method.
2. Use of UIDs to identify the internal data structures.
3. Support for an extended data structure.
4. Basic version control.

5. An XQuery querying mechanism.
6. An administrative client.
7. Backup and restoration capabilities through the administrative client.

Figure: 7.16 is a depiction of the resulting .getmore system after the second iteration. All persistence methods are present and available for usage. The Embedded and Server methods are depicted using the same Database, which maintains a single set of data manipulated through both persistence methods.

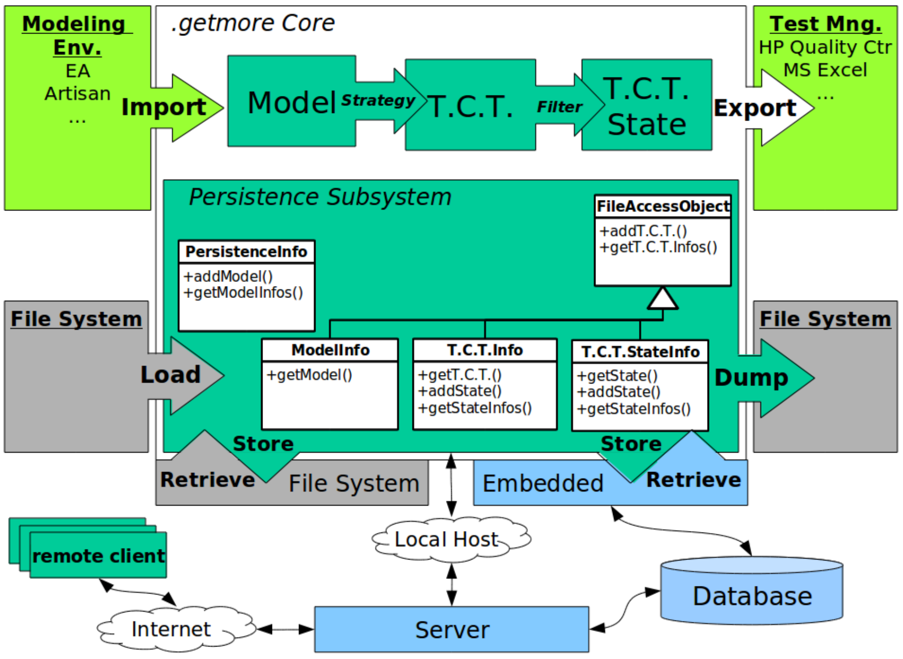


**Figure 7.16:** Resulting .getmore System – Iteration Two

## 7.13 Summary

In this chapter, the developmental effort of the second iteration was described. The iterations scope, design phases and implementation were described. The second iteration represented a more aggressive set of goals but was primarily judged for success according to the accomplishment of the project's second goal: provide local and remote access for multiple users to a communal set of data. The integration of the file system, embedded database, local database server and remote database server persistence methods was the overarching concern of this iteration. The conversion, loading and dumping of Models in the database was refactored. Support for the next version of the .getmore system was achieved through the use of UIDs and extending the internal data structure. A Model versioning system was also designed and implemented. Additionally, the groundwork for a future database filtering mechanism through XQuery was implemented.

# 8 Project Evaluation

## 8.1 Software Metrics

During this project, 14 of the 16 requirements were satisfactorily accomplished. Over 10,000 lines of code were produced. Half of which was a supporting set of 165 tests that provided over 70% line and branch coverage. The implemented code had a McCabe's cyclomatic complexity of 3.8. The tests were written using the JUnit testing framework and the statistical analysis was provided by the Cobertura tool[1]. Line coverage refers to the number of lines that were executed at least once by the tests. Branch coverage refers to the number of paths or alternatives that were chosen for conditionally executing blocks of statements. McCabe's cyclomatic complexity provides a measure of the analyzed code's complexity based on its looping pattern. The complexity can be measured by counting the number of separate "areas" created by the code's control flow diagram and adding one to that number. Alternatively, one may count the number of edges, subtract the number of nodes and add two to the result to attain the measure of complexity.

A McCabe's cyclomatic complexity of 10 or less is recommended [WM96] and a target goal of 85% is considered well tested [Kos08]. The uncovered code can be attributed to "getters" and "setters" as well as the exception catching and passing coding-style requirements of the .getmore system. Each exception that could possibly occur in the database must be caught and translated into one of the expected .getmore exception classes. This code is difficult to test, without using mock objects, and not worth the effort, since it is very simple code. Based on these metrics, the software produced is acceptably tested at the unit or module level and has a low level of complexity.

1 Cobertura Homepage: `http://cobertura.sourceforge.net`

## 8.2 Performance Evaluation

Performance evaluation was accomplished by comparing the time required for the differing methods to accomplish specified tasks. The tested tasks are: Store a Model, Retrieve a Model, Delete a Model, Get a Model's Name and Set a Model's Name. These tests were executed 10 times to provided sufficient results for the analysis. The test's mean, standard error, standard deviation and norms were calculated from the resulting times. Additionally, a relative speed factor provides a measure for comparison between the methods in each given task.

The file system and embedded database persistence methods were compared in order to evaluate the performance of serial access provided by the file system processing methods against the non-sequential access provided by the database. The systems were first tested by adding, retrieving and removing 100 Models. Next the systems were tested by getting and setting 100 Model's display name 100 times. The first set of tests all sequentially process the files, whereas the second set of tests compares the file system's sequential processing to the database's non-sequential processing of the files using XQuery and eXist-db's XQuery Update Extensions.

The results are described first by the mean, and standard error. The standard deviation of the 10 tests is provided in parenthesis and the observed norms are presented in the square brackets. Due to journaling and synchronization of the journal during the database's testing, there was typically a single outlier that dramatically increased the resulting mean. The outlier was typically a factor 10 of the other values. After removing the outlier, the mean approximated the value in the square brackets. Next the results were compared to provide a relative speed factor observing the difference between the two persistence methods. This analysis is critical in providing a groundwork and recommendation for the implementation of a filtering mechanism. Table: 8.1 on the facing page presents the performance comparison between the file system and embedded database implementations.

It should be noted that this is not a perfect comparison test, but does support a generalization of expected results. Most importantly the `java.io.File` class does not guarantee that the files are actually written to the hard drive while the embedded database guarantees the durability of the persisted resource. The accuracy of the comparison

| Test | File System | Embedded Database | Factors |
|---|---|---|---|
| **addModel()** | 63ms ±0.46(1.44)[60ms] | 101ms ±0.11(0.33)[90ms] | x1.5 |
| **getModel()** | 35ms ±0.07(0.22)[30ms] | 127ms ±0.70(2.22)[100ms] | x3.3 |
| **remove()** | 76ms ±0.14(0.44)[70ms] | 322ms ±1.65(5.22)[250ms] | x3.5 |
| **getName()x100** | 0ms ±0.00(0.00)[0ms] | 0ms ±0.00(0.00)[0ms] | x1 |
| **setName()x100** | 129ms ±0.00(0.00)[129ms] | 81ms ±0.05(0.95)[80ms] | x0.63 |

**Table 8.1:** Performance Comparision Results

could be improved by forcing the file to be flushed, which would then wait until the file is written to disk before continuing. However, due to the refactoring effort to force flushing operations for all file operations, it was deemed as not necessary.

As expected, the file system performance was much better in the sequential operations of reading and writing the Models. However, the database shows better performance when it is allowed to access and manipulate the data non-sequentially. The `.metadata` file is relatively short, only about 20 lines, and the benefits would be expected to be greater on larger files. This supports the belief that applying XQuery to filter the data will provide a performance gain for the system.

## 8.3 Developmental Evaluation

The developmental effort of this project was executed in a deliberate and thought-through manner. The various aspects of the project were considered from multiple perspectives and the solution estimated to be the better solution was chosen. A good balance between the deliberation of design and functional implementation was achieved. The integration of all persistence methods into the system in a mutually supportive manner manipulating a common set of data is a good example of this effort. It dramatically simplified the implementation effort while simultaneously exceeding the requirements. A clear understanding of the requirements and their purposes assisted the distribution of effort. The primary object was accomplished in each of the iterations and then the further requirements were accomplished according to their priority, purpose and available time. The Model version control system is a good example of this understanding. The system provides the necessary versioning functionality but was also designed with the expectation for further development. The requirement's intent was to develop a functional proof of concept not a full-fledged version control system, which is beyond the scope of this project.

The success of the developmental effort is largely due to the effective use of the modified agile approach. The weekly meetings facilitated the specification and understanding of each requirement. The monthly iterations ensured that the projects development was on track and prioritized the future developmental effort. At the lowest level, TDD ensured that a high quality product was produced.

## 8.4 Discussion and Future Work

Like most software projects, there is no final end state of the .getmore system. The system is constantly being improved with extended functionality or support for new technologies. This project accomplished the goals that it set forth to accomplish, but during the developmental effort a number of new improvements for the .getmore system were identified. It is important to provide a recommendation for future development. These recommendations will provide a starting point from which the assigned developer may orient. These recommendations fall into two categories. The first category consists of those requirements not implemented in this project. Two requirements were not fulfilled in this project. No filtering mechanism was implemented and explicit transactional support was not accessible at the .getmore system's level. The second category consists of recommendations for improvement of the current product.

### 8.4.1 Filtering Mechanism

The filtering mechanism will most likely depend on the use of XQuery statements and the XQuery support provided by the eXist-db. The XQueryTemplate class provides an example of how the functionality may be accessed and used. The compiling of often-used queries will also improve performance by reducing the query parsing effort. In order to support efficient querying, the eXist-db uses indices. Depending on the elements being filtered, the appropriate indices should be established and registered in the database's configuration. The use of compiled queries against indexed data should provide a performance boost for filtering the Test Case Trees.

### 8.4.2 Transactional Support

Transactions in the eXist-db are automatically created by the various APIs and are written to a journal in order to provide automatic crash recovery. This limits the functionality to that needed for crash recovery and is not directly usable by application code. However, transactions may be used through low-level access to the eXist-db. The eXist-db's BrokerPool provides access to a singleton TransactionManager. The TransactionManager provides the ability to begin, checkpoint, abort and commit a transaction. It is also reasonable to expect that this service will be supported in a future version of the eXist-db.

### 8.4.3 Version Control Subsystem

A version control system represents the effort of a complete project in its own right. The versioning module in the eXist-db provides significant insight into the further expansion of the current system. The version control functionality is currently provided at the Model level. The version control system could be extended to provide fine-grained support for Test Case Trees and Test Case Tree States. This will require extending histories to all elements. The current version also maintains complete copies of the Model for each committed state. It would be more efficient to maintain only the differential representing the Model's changes. The information to determine which files have been changed is available through the changed log, but the functionality must be implemented. Finally, moving the trigger's configuration file higher in the hierarchy may improve its reliability. However, it will require significantly more effort in filtering within the XQuery trigger in order to determine which Models are actually being affected. Finally, the version control is expected to grow significantly with its extension and thus should be transformed into a separate subcomponent of the persistence subsystem.

### 8.4.4 Other Areas

There are also a number of smaller concerns that could be easily improved.

- The file system persistence system will fail if the appropriate encoding is not identified in an XML file's prolog. Prologs should be added to the file or compensated for by the XML processing subcomponent.

- The `.version` file is not an XML file and is processed through a complete different system. In order to maintain consistency, the `.version` file should also be an XML file.
- The #getExpanded() and #setExpanded() methods should be deprecated in the IFileAccessObject, as the functionality should be supported through Eclipse preferences not the persistence subsystem.
- The sequential numbering option for loaded Model naming conflict should be deprecated. The Models are identified by UIDs, which should be immutable.
- The concerns of Model loading and conversion should be separated.

## 8.5 Summary

In this chapter the project was evaluated based on software metrics, the performance of the code, the execution of the developmental methodology and recommendations for further development. The code was well tested through its 165 unit test and exhibited a low to moderate level of complexity. The database implementation demonstrated poorer performance in serial processing operations but superior performance in processing that may be accomplished through non-sequential operations. The developmental effort successfully fulfilled 14 of the 16 requirements and met the clients expectations. Finally, recommendations for implementing a filtering mechanism through XQuery, transactional control and an extended version control system were discussed.

# 9 Conclusion

The .getmore system is used to automatically generate a set of test cases fulfilling a designated strategy and then filtering this set in order to provide an efficient set of tests that fulfills a designated criterion. .getmore originally persisted the system's internal data structures as files in the file system. A number of limitations make this solution a sub-optimal persistence method. The primary goal of this project was to replace the original file system persistence method with a database implementation. The secondary goal of the project was to extend the database implementation in order to provide local and remote access to multiple users for a communal set of data. This is to allow an entire team of software testers to work in unison to test the system.

The legacy .getmore system was first analyzed to determine the coupling between the persistence subsystem and the file system and an appropriate NXD, eXist-db, was identified. In the first iteration, the access to the eXist-db and a façade to the XML:DB API was encapsulated in a StorageComponent class. Next new implementations of the persistence subsystem's DAOs, using the new encapsulated database functionality instead of the file system, were created. Effectively replacing the file system dependant persistence subsystem with a database implementation. After the first iteration the .getmore system persisted its internal data structures in an embedded eXist-db through an XML:DB API façade described in the StorageComponent class.

In the second iteration, the persistence subsystem was extended to support multiple modes of persistence. The file system and embedded database persistence systems were maintained while methods for accessing a database server, either locally or remotely, were provided. Additionally, the database embedded and server implementations were integrated to use the same data storage location so that a user operated on the same set of data in both embedded and local database modes. Finally, the use of persistence properties allows the system to dynamically declare the persistence method during operation. This created a fault-tolerant persistent framework. A basic Model version control system was also implemented using database triggers. This allows a user to commit or save the current state of a Model and then revert to a previously committed state at a later point. The versioning system supports version control within the eXist-db. The .getmore system was also upgraded to support a number of other features to include the use of UID element identification, an extended internal data structure, database backup, restoration and administrative capabilities. The groundwork for an XQuery filtering mechanism was also provided.

The .getmore system is a tool for the automatic generation of coverage test cases. This project successfully improved the persistence subsystem's fault-tolerance, durability and flexibility. The persistence subsystem may be employed in file system, embedded database, local database server or remote database server modes. All database modes are integrated to operate on a single set of data and the database server modes provide local and remote access for multiple users.

# A Project Requirements Purpose Analysis

In this appendix the requirements management process will be described and then each requirement will be restated and its purpose will be analyzed.

## A.1 R1-4: Persist Internal Data Structures in an Embedded Database

These are the key requirements of this project. The storage of the internal data structures within an embedded database will alleviate the .getmore system from its dependency upon the file system's storage capabilities and limitations. The intent of this requirement is to:

1. Transparently persist and retrieve the data within an embedded database, effectively replacing the use of the file system persistence subsystem.
2. Prevent undesired tampering of the persisted data.
3. Ensure the data remains uncorrupted.
4. Collect the data in a single secure location.

Provide a basis for the implementation of the other requirements.

*See Section 3.1.1 on page 7 for the description of the requirement.*

## A.2 R5: Provide Local and Remote Access for Multiple Users

The distributed multiple user access to the persisted data is intended to provide a team of tester the ability to cooperate and manipulate a single set of generated tests. This will increase the end-user's efficiency and the system's usefulness.

*See Section 3.1.2 on page 8 for the description of the requirement.*

## A.3 R6: Provide Filtered Access to Persisted Data

The system will take advantage of the database's querying mechanisms to reduce the amount of data returned, which in-turn will improve the processing performance of the filtering strategies.

*See Section 3.1.3 on page 9 for the description of the requirement.*

## A.4 R7-16: Extended Requirements

**R7: Provide Model Versioning Control.** The intent of this requirement is to provide the user the ability to experiment with new ideas without the fear of ruining the current state of work. It provides a similar benefit as a configuration management system such as a CVS or SVN.

*See Section 3.1.4 on page 9 for the description of the requirement.*

**R8: Integrate the Database into .getmore's Installation Process.** The intent of this requirement is to provide the end-user a functional product without the need for additional support in order to configure and start using the system. The automatic self-installation process reduces the effort required by the end-user and developmental team alike.

*See Section 3.1.4 on page 9 for the description of the requirement.*

**R9: Convert Models Created by other .getmore Versions.** The purpose is to allow the .getmore system to continue its evolution while maintaining a backward/-forward compatibility for the models being manipulated by the end-user. This makes the upgrading process as simple and transparent to the end-user as possible.

*See Section 3.1.4 on page 9 for the description of the requirement.*

**R10: Dump Persisted Models to File System.** The intent of this requirement is to provide a simple method for data backup, data transfer between .getmore systems and the employment other file based operations (such as schema version conversion) on the persisted data. It also provides the ability for the user to physically open the files and visually inspect their contents for correctness.

*See Section 3.1.4 on page 10 for the description of the requirement.*

**R11: Load Models from File System into the Database.** The intent of this requirement is to provide a simple method for data recovery, data transfer between .getmore systems and the employment other file based operations (such as version conversion) on the persisted data.

*See Section 3.1.4 on page 10 for the description of the requirement.*

**R12: Provide Transactional Support.** The intent of this requirement is to provide the .getmore system with an assurance that the data has been correctly persisted and the ability to abort any transactions determined to be not appropriate.

*See Section 3.1.4 on page 10 for the description of the requirement.*

**R13: Provide Backup and Restoration of Persisted Data.** The backing up and restoring of data from the backup provides an additional level of security and stability to the system and the user. It will allow the user to "roll-back" the data set to a previous state or recover from a catastrophic loss of data in the database.

*See Section 3.1.4 on page 10 for the description of the requirement.*

**R14: Provide a Database Administrative Client.** The administrative client provides another access to the data for direct manipulation and recovering for unexpected events. This provides additional stability as well as another avenue of recovery beyond deleting all of the data in the database and starting over.

*See Section 3.1.4 on page 10 for the description of the requirement.*

**R15: Support UIDs for Element Identification.** The use of unique identification strings will provide additional stability to the system and allow for a simpler manipulation of the data structures names. Thus the path to a specified data structure becomes independent of the names of its superior data structures.

*See Section 3.1.4 on page 11 for the description of the requirement.*

**R16: Support an Extended Internal Data Structure.** This will support a new .getmore system and its new internal data structure. This new structure will provide the additional flexibility for further expansion of the .getmore system.

*See Section 3.1.4 on page 11 for the description of the requirement.*

**RX: Provide Future Developmental Recommendations.** The intent of this requirement is to provide a basis for follow-on developmental efforts, either at the Friedrich Alexander University or SeppMed GmbH internally.

*See Section 3.1.4 on page 11 for the description of the requirement.*

# Bibliography

[BBC$^+$98] Jon Bosak, Tim Bray, Dan Connely, Eve Maler, Gavin Nicol, Lauren Sperberg-McQueen, C. Micheal Wood, and James Clark. W3C XML Specification DTD ("XMLspec"). Technical report, ArborTech Inc., `http://www.w3.org/xml/1998/06/xmlspec-report-19980910.htm`, September 1998.

[BCF$^+$07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Technical report, XML Core Working Group, `http://www.w3.org/TR/xquery/`, January 2007.

[Bec99] Kent Beck. *eXtreme Programming eXplained.* Addison–Wesley, 1999.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. III McCormick, and Thomas J. Mowbray. *AntiPatterns – Refactoring Software Architectures and Projects in Crisis*, pages 133–137,167–174. Wiley Computer Publishing, 1998.

[BPSM$^+$08] Tim Bray, Jean Paoli, C. Micheal Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report, XML Core Working Group, `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008.

[CC09] Larry Cable and Thorick Chow. JSR 173: Streaming API for XML. Technical report, Java Community Process, `http://jcp.org/en/jsr/detail?id=173`, July 2009.

[CD99] James Clark and Steve Derose. XML Path Language (XPath) Version 1.0. Technical report, XML Core Working Group, `http://www.w3.org/TR/xpath/`, November 1999.

[CFL$^+$99] Kurt Cagle, Mark Fussell, Nalleli Lopez, Dan Maharry, and Saran Rogerio. *XQuery Early Adopter*. Addison–Wesley, 1999.

[eXi10] eXist. eXist-db: Open Source Native XML Database. `http://exist.sourceforge.net`, 2010.

[FW04] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer. Technical report, XML Core Working Group, `http://www.w3.org/XML/Schema`, October 2004.

[GHJV95] Erich Gamma, Richard Helm, Richard Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 17–18, 185–193,325–330. Addison–Wesley, 1995.

[Grü10] Christian Grün. BaseX Homepage. `http://www.inf.uni-konstanz.de/dbis/basex/`, 2010.

[ISO86] ISO. ISO8879:1986 - Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML), 1986.

[ISO08] ISO. ISO/IEC9075-14:2008 - Information Technology - Database Languages - SQL - Part 14: XML-Related Specifications (SQL/XML), 2008.

[Kat04] Howard Katz, editor. *XQuery from the Experts*. Addison–Wesley, 2004. Chamberlin, Don and Draper, Denise and Fernández, Mary and Kay, Michael and Robie, Jonathan and Rys, Michael and Siméon, Jérôme and Tivy, Jim and Wadler, Philip.

[Kos08] Lasse Koskela. *Test Driven: Pratical TDD and Acceptance TDD for Java Developers*. Manning Publications Co., 2008.

[Lar04] Craig Larman. *Agile & Iterative Development - A Manager's Guide*. Addison–Wesley, 2004.

[Le 04] Philip Le Hégartet. Document Object Model (DOM) Technical Reports. Technical report, W3C Core Working Group, `http://www.w3.org/DOM/DOMTR`, July 2004.

[Meg05] David Megginson. SAX Homepage. `http://www.saxproject.org/`, May 2005.

[MP05] Irena Mlynkova and Jaroslav Pokorny. XML in the World of (Object-) Relational Database Systems. In Olegas Vasilecas, Wita Wojtkowski, Jože Zupančič, Albertas Caplinskas, W. Wojtkowski, and Stanisław Wrycza, editors, *Information Systems Development*, pages 63–76. Springer US, 2005.

[MyS10] MySQL. MySQL: The world's most popular open source database. `http://www.mysql.com`, 2010.

[PCSF08] C. Micheal Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion.* O'Reilly Media Inc., 2008.

[Sar06] Poornachandra Sarang. *Pro Apache XML.* Apress, 2006.

[Sch95] Ken Schwaber. SCRUM Development Process. In *OOPSLA '95 Workshop on Business Object Design and Implementation*, 1995.

[Sed10] Sedna. Sedna: Native XML Database System. `http://www.modis.ispras.ru/sedna`, 2010.

[SX01] Kimbro Staken and XML:DB API MAILING LIST, Members. XML Database API Draft. `http://xmldb-org.sourceforge.net/xapi/xapi-draft.html`, Sep 2001.

[TLMG10] Peter Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action.* Manning Publications Co., 2nd edition, 2010.

[VV06] Ajay Vohra and Deepak Vohra. *Pro XML Development with Java Technology.* Apress, 2006.

[WM96] Arthur H. Watson and Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical report, National Institute of Standards and Technology, September 1996.

[XLWB09] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In *Proceedings of the 35th*

*SIGMOD international conference on Management of data*, SIGMOD '09, pages 719–730, New York, NY, USA, 2009. ACM.