Technische Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h. c. Theo Härder

A Java Content Repository backed by the native XML Database System XTC

JSR 170 compliant implementation

Diploma Thesis

submitted by
Sebastian Prehn

Betreuer:
Prof. Dr.-Ing. Dr. h. c. Theo Härder
Dipl.-Inf. Karsten Schmidt

Day of issue: 21. Feb. 2008
Day of delivery: 31. Jul. 2008

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit mit dem Thema “A Java Content Repository backed by the native XML Database System XTC” selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Hereby I declare that I have self-dependently composed the Diploma Thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

Kaiserslautern, 31. Jul. 2008

Sebastian Prehn

Abstract

JSR 170 spezifiziert die *Java Content Repository* (JCR) Schnittstelle. Diese Schnittstelle wird als Standard im Bereich *Web-Anwendungen* und *Content Management* akzeptiert. Sie gliedert sich in *Level 1* (lesender Zugriff) and *Level 2* (Lese- und Schreibzugriff) und beschreibt darüber hinaus vier weitere optionale Funktionen. Das in JSR 170 beschriebene hierarchische Datenmodell weist starke Analogie zu XML auf. Jedoch verwenden die meisten JCR-Implementierungen relationale Datenbanken. Durch native XML Datenbanken, wie XTC, können XML-Daten effizient verwaltet werden. Diese Arbeit beschreibt das Design und die Implementierung eines *Level 2* JCRs, welches alle Anforderungen an die Persistenz mit Hilfe von DOM und XQuery Operationen auf XML-Dokumenten in XTC erfüllt. Die optionalen Funktionen "Versionierung" und "Transaktionen" werden ebenfalls unterstützt. Um die Implementierung zu testen werden zwei Demo-Anwendungen (Blog & Wiki) entwickelt und Vergleichstests gegen die Referenzimplementierung angestellt.

JSR 170 specifies the *Java Content Repository* (JCR) interface. This interface is accepted as a standard in the field of *Web Applications* and *Content Management*. The specification is structured in *Level 1* (read-only access) and *Level 2* (read and write access). Furthermore, it specifies four optional features. The hierarchic data model described in JSR 170 exhibits strong analogy to XML. However, most JCR implementations use relational database systems. Native XML databases, such as XTC, are able to manage XML data efficiently. This thesis describes the design and implementation of a JCR which meets all requirements on persistence employing DOM and XQuery operations on XML documents in XTC. Optional features "Versioning" and "Transactions" are supported. In order to test the implementation two demo applications (blog & wiki) are developed and benchmarks are run against the reference implementation.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Use Case: CoffeeBeen Inc.

A typical commercial website—a small use case to begin with:

CoffeeBeen Inc. sells Coffee online. On their website the company advertises itself and its products. Current news are published in a blog system. Customer feedback gets recorded in form of comments on the product pages and blog entries. The company's customer care department services a FAQ section. The public relations experts author the rest of company's online presentation.

The website is managed by a *Content Management System* (CMS), more precisely, a *Web Content Management System* (WCMS). As the name suggests, a WCMS is responsible to manage content (files, images, articles etc.) and to present it in form of web pages (see figure 1.1).

The software allows multiple users to colaborate concurrently on the web pages. The users create or modify content elements. On the web front-end the content elements are merged into the website layout and presented to the users online.

The requirements on a typical WCMS include full-text search, versioning and a mechanism to handle fine grained concurrent read and write access on the content. These are needs, a regular filesystem does not sufficiently support. Therefore, the WCMS requires a system on top of the pure data storage to handle these common requirements. This system is usually referred to as a *Content Repository*.

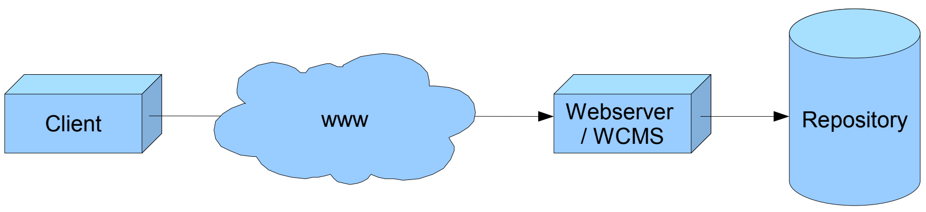


Figure 1.1: Typical setup for WCMS systems

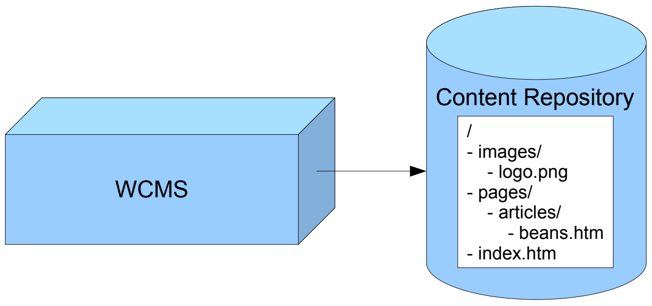


Figure 1.2: *Content Repository* as hierarchical storage back-end to a WCMS

1.2 Content Repository

A *Content Repository* (CR) is a hierarchical data store for content. This content can be anything from primitive datatypes, texts, image files or other binary documents along with accompanying meta data (see figure 1.2).

While all *Content Management Systems* must provide some sort of *Content Repository* implementation to store their data in[1], it is unclear what such a repository must feature.

David Nuescheler defines a *Content Repository* as follows: "A *Content Repository* is a high-level information management system that is a superset of traditional data repositories. A *Content Repository* implements *content services* such as: *author based versioning*, *full textual searching*, *fine grained access control*, *content categorization* and *content event monitoring*. It is these *content services* that differentiate a *Content Repository* from a *Data Repository*. [Nue06]"

1.3 Java Content Repository

CoffeeBeen Inc., of the initial use case (see chapter 1.1), desires to integrate their web presentation with other applications of their IT infrastructure. They need to export the product catalog from their Enterprise Resource Planning (ERP) software onto the web page. In addition, employees querying the company's knowledge management system should also find matches in the online user comments and FAQ section (see figure 1.3).

It is a common requirement to integrate repository content with other applications. *Content Repositories* typically provide an interface for applications to query and modify the underlying content. There are many CR solutions on the market. Each one offering its own API to interact with content. Proprietary APIs limit the compatibility to ready-made, vendor specific solutions. They tightly couple the CR and the integration partner or render the integration endeavor impossible. Open, but non-standardized APIs, may require

[1] For the sake of simplicity the WCMS and *Content Repository* are viewed as separate components. In practice the repository layer might not clearly be separated from the WCMS. Instead, it might be considered to be part of the system, querying an external database. A *Content Repository*, however, is more than just a database or the file system.

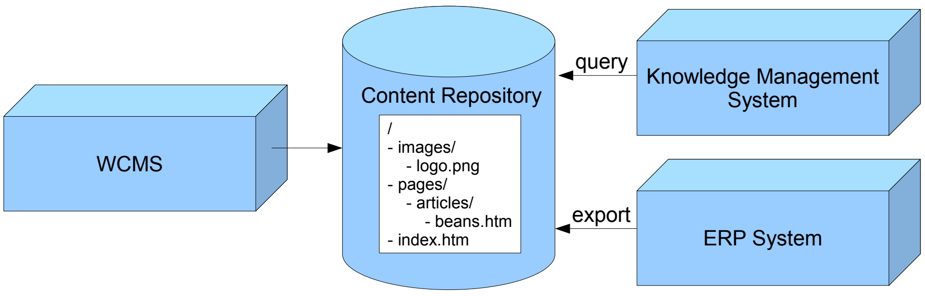


Figure 1.3: *Content Repository* integration with other systems

custom-made adapter software between each of the integration partners. The integrators therefore need to conquer the diverse APIs. Application vendors need to adapt their products to every single API. In summary, the chaos of APIs makes integration costly in terms of labor, time, and money.

The obvious solution to the integration problem is a common, adequate, and open interface for content repositories. In an effort to unify the *Content Repository* APIs in the Java world a common interface was specified in the *Java Community Process* (JCP) under the name *Content Repository API for Java Technology* JSR[2] 170. A *Java Content Repository* (JCR) is a CR implementation that complies to the JSR 170 specification [3].

The common interface enables reuse, exchangeability and interchangeability of the repository layer. It unifies the diverse requirements in the *Content Repository* market in a set of mandatory and optional APIs. The specification is widely accepted and this suggests that the JSR 170 standardization might lead to a similar unification observed subsequently to the introduction of the SQL standard: Today nobody would build a proprietary query language for a relational database system. [NN04]

With JSR 170 (web) application developers can leverage the power of ready-made *Content Repository* solutions interacting with a single open API and without committing themselves to a certain repository implementation.

The customer is not bound to a certain vendor. This leaves a choice to select either a best-of-breed solution or some other product that integrates well into the company's IT infrastructure. The system stays open for integration with other applications.

Several WCMSs already use JCR repositories, e.g. Magnolia and Alfresco. Even non-Java WCMS Systems are not excluded. The widespread PHP based open-source content management system *TYPO 3* has published plans to switch to a JCR in version 5.0 [Dam07].

Java Content Repositories are not limited to be used in the context of Web Content Management only. Any form of CMS, e.g. *Enterprise Content Management Systems*, or any other Java based application can easily utilize the power of JSR 170 compliant implementations as a feature rich storage back-end.

[2] Java Specification Request

[3] see chapter 2 for a more detailed description of JSR 170

There are already several implementations of the JSR 170 pseudo-standard. Here a list of known implementations, without claiming completeness.

Open-source implementations:

- Apache Jackrabbit[4]—Reference Implementation (RI)
- Toshiro JCR
- Jeceira
- Alfresco

Commercial products:

- Content Repository Extreme (CRX) of Day Software AG
- Oracle Beehive Java Content Repository API[5]

1.4 Native XML Storage

The way a JCR stores data permanently is not predefined by the specification. The reference implementation, e.g., comes with several implementations for their *persistence storage* layer. The different implementations allow the reference implementation to be backed e.g. by a relational database or simple XML files. Doubtless to say, the most prominent approach for production use is the mapping to relational databases. This is due to the wide availability of powerful relational database systems.

However, mapping the hierarchy of the JCR content tree into a hierarchical format, more precise XML [BPS00], seems to be the most natural, straight forward approach. The specification even relies on XML as in- and export format for the complete repository. This shows that the expressiveness of XML fits JCR content very well.

The use of simple XML files in the filesystem is not acceptable[6]. With pure XML files no fine grained concurrent access would be possible. It is impossible to guarantee the ACID properties or only at the cost of locking the whole file, eliminating any concurrent access.

Since the requirements on the JCR include typical database requirements, it makes sense to manage the XML data in a database system as well. In order to store JCR data in such a database, the system must be capable of handling XML data and support at least the following features:

- in document (subtree) modifications (update, delete)
- in document fine grained transactional control

[4] `http://jackrabbit.apache.org`

[5] `http://www.oracle.com/technology/products/beehive/examples/jcr.html`

[6] Jackrabbit's XML file persistence store is not recommended for production use due to the lack of performance.

- XPath and XQuery interface (problem: standardization of interface)

This already outrules several XML-enabled database systems that can only store whole XML files as unstructured text values as their finest granulate or cannot modify subtrees.

In XML-enabled databases where XML data is "shredded" into tables, structural information in the tree-based schema is modeled by joins between tables in the relational schema. XML queries are converted into SQL queries over the relational tables, and even simple XML queries often get translated into expensive sequences of joins in the underlying relational database. [JAKC$^+$02]

In contrast, specially tailored XML database systems potentially provide efficient data structures and indexes, efficient handling of XML queries, and support for sophisticated transaction handling on XML documents.

The upcoming solution for XML storages are therefore native XML database systems. These are systems designed from scratch, that do not internaly map XML to relational tables or object structures, but implement real tree data structures and corresponding query operators on these structures.

Overview native XML database systems, without claiming completeness:

eXist open-source database management system entirely built on XML technology[7]

Oracle Berkeley DB XML open-source XML database with XQuery-based access[8]

MonetDB open-source database system for high-performance applications in data mining, OLAP, GIS, XML Query, text and multimedia retrieval[9]

Natix [FHK$^+$02] persistent XML storage, including high-performance document import and export, access via DOM [DOM] and SAX [SAX] interfaces, transaction processing with recovery, and scalable evaluation of XPath 1.0 [CD99] queries.

Sedna open-source native XML database[10]

Tamino [Sch01] commercial native XML database system of Software AG.

Timber [JAKC$^+$02] open-source database system based on TAX (Tree Algebra for XML) that is for manipulating XML data in form of forests of labeled ordered trees[11].

XTC [HH07] closed-source native XML database system of AG DBIS TU Kaiserslautern.

7 `http://exist.sourceforge.net`

8 `http://www.oracle.com/database/berkeley-db/xml/index.html`

9 `http://monetdb.cwi.nl/`

10 `http://modis.ispras.ru/sedna`

11 `http://www.eecs.umich.edu/db/timber`

Our research group has developed the *XML Transaction Coordinator* (XTC) as native XML database prototype system [HH07]. XTC seems suitable to support a JCR to XML mapping efficiently.

The JCR implementation is expected to gain performance through XTC's native tree data structures (including its powerful labeling schemes), physical operators, and optimized transaction handling. Thus, this JCR implementation will be implemented as an additional layer on top of XTC enhanceing the system by a JCR interface. Meanwhile XTC's concepts are put to the test through a real-life application.

Chapter 2

JCR Specification

2.1 Java Specification Request 170

JSR 170: Content Repository for Java^{TM}technology API is a Java Specification Request (JSR)[1] lead by David Nuescheler of Day Software AG. His effort started February 2002 and the specification was finally released on 17. June 2005. The latest maintenance release version 1.0.1 dates back to 24. April 2006, the version that is used in this work.

The specification is structured in two compliance levels (*Level 1* and *Level 2*) and describes four optional features (Versioning, Transactions, Locking, Observation). *Level 1* is mandatory for every JCR. In order to find out what levels and features a repository supports, the repository can be queried (see the following code listing 2.1 and the output given in table 2.1).

```
for(String key : repository.getDescriptorKeys()){
  System.out.println(key + ":\t" + repository.
       getDescriptor(key));
}
```

Listing 2.1: Querying repository features

This chapter will not delve into details, but will give an overview of the specification along with a basic example. Please refer to the specification document [Nue06] for more information.

2.2 JCR Model

A JCR accommodates one or more workspaces. A workspace has a name and represents a tree of items (see figure 2.1). The JSR 170 specification models this tree according to the GoF[2] Composite Design Pattern [GHJV95]. An item

[1] JSR 170 in Java Community Process (JCP) `http://www.jcp.org`

[2] Gang of Four

Key	Value
query.xpath.doc.order:	true
query.xpath.pos.index:	true
level.1.supported:	true
level.2.supported:	true
jcr.specification.version:	1.0
jcr.repository.vendor.url:	http://wwwlgis.informatik.uni-kl.de
jcr.specification.name:	Content Repository for Java Technology API
jcr.repository.name:	XTC JCR
jcr.repository.vendor:	AG DBIS, TU Kaiserslautern, Germany
jcr.repository.version:	0.1
option.versioning.supported:	true
option.query.sql.supported:	false
option.transactions.supported:	true
option.locking.supported:	false
option.observation.supported:	false

Table 2.1: Repository Descriptor of XTC JCR

is either a node or a property (see figure 2.2). Each node has a name and an arbitrary number of child items. Each property has a name and additionally stores values. So in summary: Workspaces and nodes structure the repository while properties store the data.

JCR nodes and properties are typed. A property must be of a primitive type: BOOLEAN, DOUBLE, LONG, STRING, DATE, BINARY, or REFERENCE. Each node must be of a primary type and can have several mixin types. A node type (primary or mixin) defines which child items must exist, are allowed to exist, or are automatically created. It defines, furthermore, which child items are user-editable and whether the node can have same named siblings as child nodes. For more details on the type system see the specification [Nue06] and chapter 4.2.3.

Figure 2.3 shows an example of such a workspace content tree. The root node is the only node that has an empty name. All other nodes carry names. Note that all these names in this example are prefixed with “cb:”. This arbitrarily chosen prefix stands for *CoffeeBean*, the name of the imaginary company. Custom prefixes allow to separate namespaces in JCR, analogous to XML namespaces. As previously said, nodes and properties are typed. The nodes in this example could be of type “nt:unstructured”, the least restrictive and default node type. In this example you can, furthermore, observe properties of type STRING and DATE.

2.3 Level 1

Level 1 specifies a *Java Content Repository* with read access. This enables storage systems to expose their content through the standardized JCR interface. In order to keep the barrier low, the *Level 1* specification is intentionally fairly easy to implement.

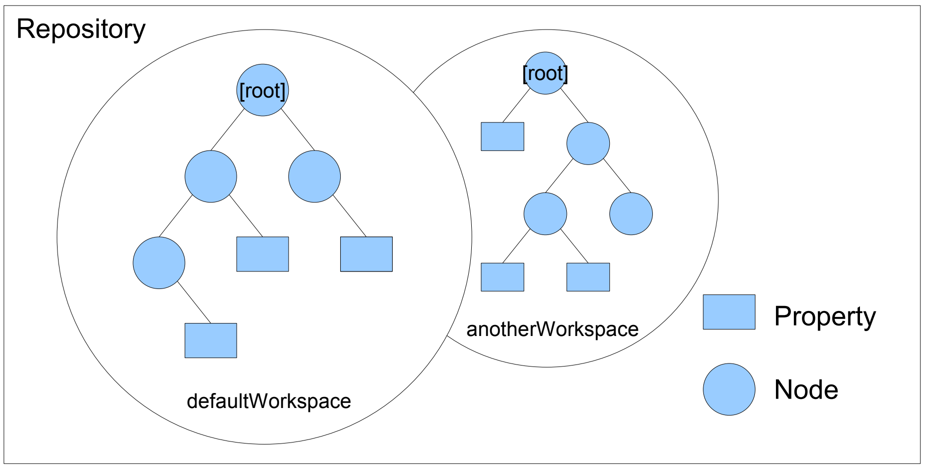


Figure 2.1: Overview of the repository structure

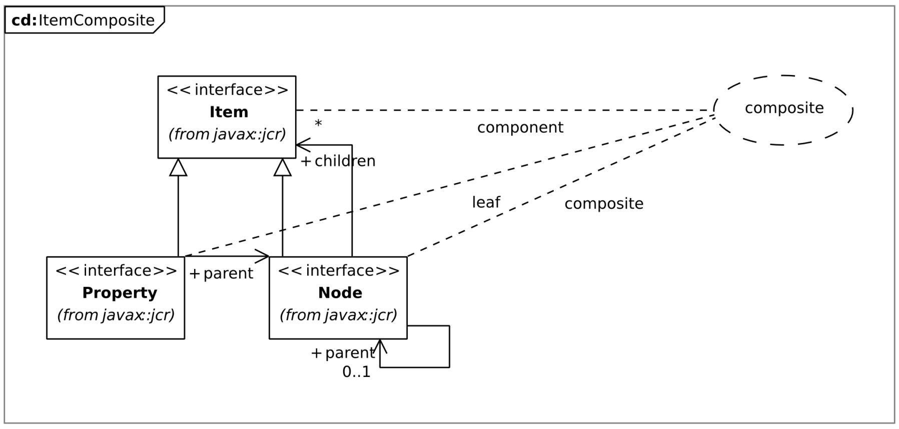


Figure 2.2: GoF Composite Design Pattern for JCR Item, Node, and Property

Level 1 of the specification requires the following features [Nue06]:.

- Retrieval and traversal of nodes and properties
- Reading the values of properties
- Remapping of transient namespaces
- Export to XML/SAX
- Query facility with XPath syntax
- Discovery of available node types
- Discovery of access control permissions

```
[root]
|— cb:pages
|   |— cb:articles
|   |   |— cb:beans
|   |   |   |— cb:title = “A couple of coffee beans every morning”
|   |   |   |— cb:date = “2008-05-13T15:39:03:010Z“
...
|   |   |— cb:milk
|   |   |   |— cb:title = “Put milk in your coffee”
|   |   |   |— cb:date = “2008-05-12T16:31:04:052Z“
...
|   |   |— cb:sugar
|   |   |   |— cb:title = “Some people like sugar in their coffee”
|   |   |   |— cb:date = “2008-04-11T06:15:12:452Z“
...
|   |   |   |— cb:comments
|   |   |   |   |— cb:comment
|   |   |   |   |   |— cb:author = “pure81”
|   |   |   |   |   |— cb:subject = “I don’t like sugar with my coffee”
|   |   |   |   |   |— cb:text = ...
|   |   |   |   |— cb:comment
|   |   |   |   |   |— cb:author = “sweety84”
|   |   |   |   |   |— cb:subject = “Sugar is a must have”
|   |   |   |   |   |— cb:text = ...
...
|
|— products
...
```

Figure 2.3: Excerpt of a JCR workspace content tree—“CoffeeBeen Inc.”

The specification comes with a set of defined Java interfaces[3]. These interfaces shield the repository client from any implementation specific details. The first three interfaces a JCR user gets in contact with are: *Repository*, *Session*, and *Workspace*.

A *Repository* instance is the first entry point into JCR. It allows to query the features of a concrete JCR implementation and to login into a workspace. How an instance of this type is obtained is not specified. A common solution is to retrieve a reference via a JNDI[4] service (see figure 2.4).

After logging into the *Repository* a client receives an object of type *Session* (see figure 2.5). An instance of type *Workspace* is accessible via the *session* object. *Workspace* and *Session* have a one to one relationship (see figure 2.6). The difference between a session and a workspace is only of importance to *Level 2* implementations. In a *Level 1* implementation, the session grants direct access to content items, i.e. nodes and properties, while the workspace instance allows to obtain a manager to run queries against the repository. That means

[3] Java package *javax.jcr.**

[4] Java Naming and Directory Interface

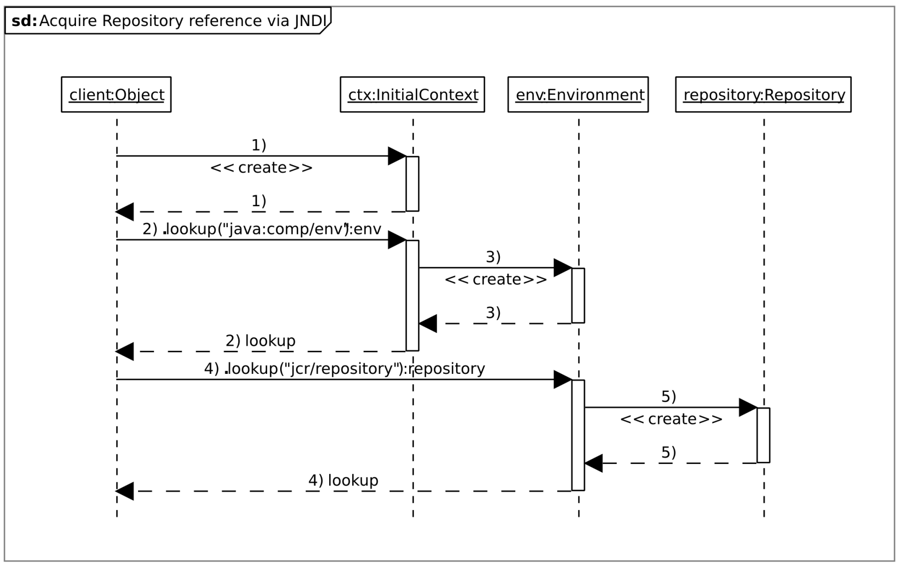


Figure 2.4: Obtaining a Repository reference via JNDI

a repository can be queried in two forms: navigational or declarative.

The navigational access via the *Session* works by directly traversing top-down from node to node and from node to property through the content tree. This form of access is used to navigate to content via known workspace structures (see listing 2.2).

```
Node root = session.getRootNode();
Node beans = root.getNode("cb:pages/cb:articles/beans");
Property title = beans.getProperty("cb:title");
System.out.println(title.getString());
```

Listing 2.2: Direct node access (workspace: see figure 2.3)

Declarative queries over the *Workspace* interface are useful when the content position in the workspace structure is unknown, i.e. for searching the repository. For example, to find all comments of user "sweety84" a JCR XPath query can be issued via the *QueryManager* (see listing 2.3).

A query specifies a subset of nodes within a workspace that meet the stated constraints. The constraints fall into three categories: [Nue06]

- Path constraint: This limits the returned nodes to certain subtrees in the workspace.
- Property constraint: This limits the returned nodes to those with particular properties having particular values.
- Type constraint: This limits the returned nodes to those with particular primary or mixin node type.

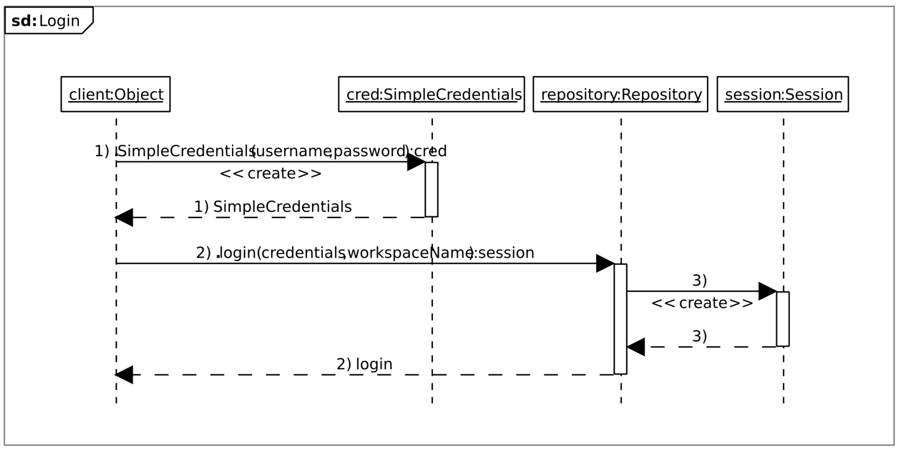


Figure 2.5: Logging into workspace “defaultWorkspace”

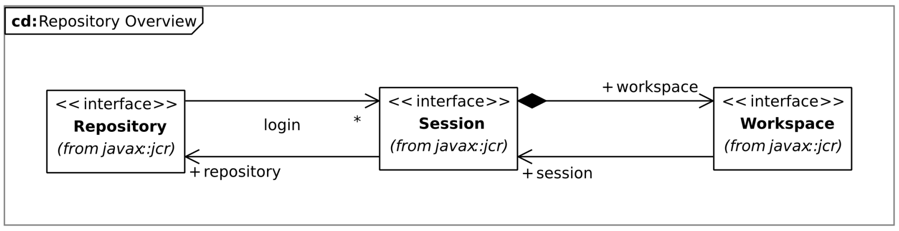


Figure 2.6: Overview: Repository, Session, Workspace

Queries can be expressed in a SQL or XPath [BBC+07] like syntax. The XPath syntax is required, while the SQL syntax is optional. In the following work we will only concentrate on XPath style queries, since SQL queries will not be supported by this implementation.

The JCR XPath query syntax is very similar to the original XPath syntax, but with a reduced complexity. In order to express path constraints three out of the 13 XPath axes[5] are supported in the location steps:

- *child* axis: abbreviated syntax “/”, the default axis
- *descendant-or-self* axis: abbreviated syntax “//”
- *attribute* axis: abbreviated syntax “@” (JCR properties are treated like XML attributes)

Only the abbreviated syntax is supported. XPath style axis selection “::” is not supported. Support for other axes is not required.

[5] XPath axes according to specification [BBC+07]: ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling, self

```java
Workspace ws = session.getWorkspace();
QueryManager qm = ws.getQueryManager();
Query q = qm.createQuery("//cb:comment[@cb:author=\"
    sweety84\"]",Query.XPATH);
QueryResult result = q.execute();

NodeIterator iterator = result.getNodes();
while(iterator.hasNode()){
  Node n = iterator.nextNode();
  System.out.println(n.getName());
}
```

Listing 2.3: Declarative query (workspace: see figure 2.3)

Property constraints are expressed as XPath predicates with the abbreviated syntax “@” for XML attributes, as in: *[@cb:author = ”sweety84”]*. Operators in a predicate are: *(=, !=, <, <=, >, >=)*. Functions that can be used are for example: *not()* and *jcr:like()*. The positional square bracket index notation of XPath is optional in a JCR, but will be supported by this implementation.

Type constraints can be expressed in a location step with the XPath function *element(elementname, typename)*. The *typename* corresponds JCR node types. The query *//element(* , nt:file)* would select all file resources with arbitrary name, for example.

2.4 Level 2

A *Level 2* implementation is a *Level 1* repository that supports the content to be modified. A new child node can easily be created:

```java
Node newNode = articles.addNode("cb:hot_summer");
newNode.setProperty("cb:text","A nice summer day.");
session.save();
```

Listing 2.4: Creating a new *Node* and setting a property

Each session keeps a virtual local copy of the repository which is called a *transient repository*. Changes to items are initially made in that *transient repository*, visible only to the current session.

These are the *Level 2* methods that write to the *transient repository*:

- *Node*: *addNode*, *setProperty*, *orderBefore*, *addMixin*, *removeMixin*
- *Property*: *setValue*
- *Item*: *remove*
- *Session*: *move*, *importXML*
- *Query*: *storeAsNode*

Modified items are transient, since the modifications are discarded when the session is closed. Within a session the changes on an item's subtree stay transient until they are made persistent by calling *save* on that item. Calling the *refresh* method discards all transient changes on the item and its subtree. The *transient repository* keeps all transient items and shadows the actual persistent items, when accessed through that session. This means the user of the session sees the repository as if the changes were already applied.

One could get the idea that *save* corresponds to a *commit* and *refresh* corresponds to *rollback* in traditional transactions. However, note that the scope of *save* or *refresh* on an item is limited to the item's subtree. Consequently, a partial *save* and a partial *refresh* of the work must be supported. This concept differs from the *all or nothing* (Atomicity) approach known from traditional transactions and inhibits a straight forward mapping to database transactions.

Chapter 3

Design & Project Setup

3.1 Two Approaches

The aim of this work is to design and implement a *Java Content Repository* on top of the native database system XTC. This is why the development name of this implementation is "XTC JCR". The first design decision to make was whether to extend an existing solution or create an implementation from scratch.

The *Apache Jackrabbit* [1] project provides the reference implementation of the JSR 170 specification. This open-source project is build in a modular fashion and supports multiple plugable storage back-ends. The first idea was to extend that implementation with an XTC specific persistence back-end. This approach has the advantage that major parts of the specification must not be reimplemented but are already supported by *Jackrabbit.* Furthermore, the reference implementation supports all optional features out of the box.

However, the persistence manager interface of the reference implementation is relatively primitive. It is operated through many levels of abstraction. Using this interface the full power of XTC could not be leveraged. This concerns mainly the native hierarchic data structure and the native XPath query support that XTC offers. The alternative approach is a complete reimplementation of JSR 170 from scratch. A newly designed implementation, ready-made specifically for XTC, promises to take full advantage of XTC's features[2].

The first major design decision in this project was made in favor of the second approach, a new implementation from scratch.

3.2 System Architecture

The XTC JCR system is implemented on top of the *XTC Server.* It connects to the server via the *XTCDriver* interface and implements the JSR 170 API

[1] `http://jackrabbit.apache.org/`

[2] As a nice side-effect a new implementation can leaverage modern Java 1.6 features.

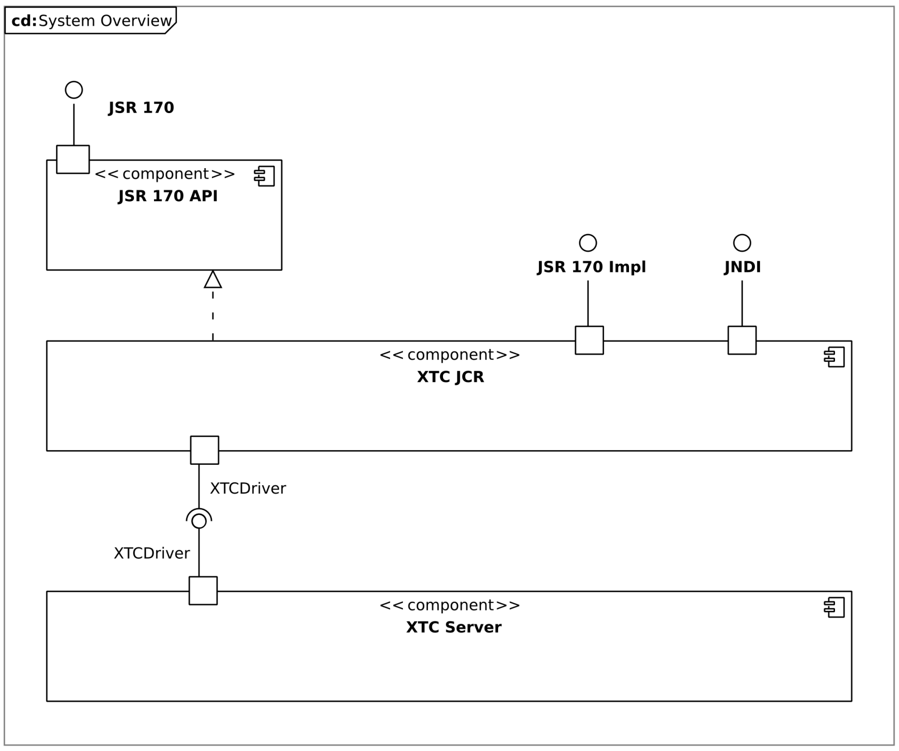


Figure 3.1: System Overview

specification (see UML[3] component diagram figure 3.1). Both *JSR 170 API* and *XTC Server* are given. The *XTC JCR* component is the target of this work.

3.2.1 Interface Description

The *JSR 170* interface stands for the standardized JCR interface described in chapter 2. This interface is fully specified in the JSR 170 specification [Nue06].

The *XTCDriver* is a proprietary interface, functionally similar to the JDBC standard for relational databases, yet specifically tailored to the needs of a native XML database system. The *XTCDriver* lets clients access XTC's *DOM RMI* and *API RMI* top-level interface services (see figure 3.2). Via *API RMI* XQuery [CBa] statements can be issued and data can be transfered in both directions as serialized XML. *DOM RMI* grants DOM based access for read and write access. The *XTCDriver* is defined by the DBIS group, but can be customized to specific XTC JCR needs.

[3] All UML diagrams in this thesis adhere to UML version 2.0 [OMG03] and recommendations given in [Oes05].

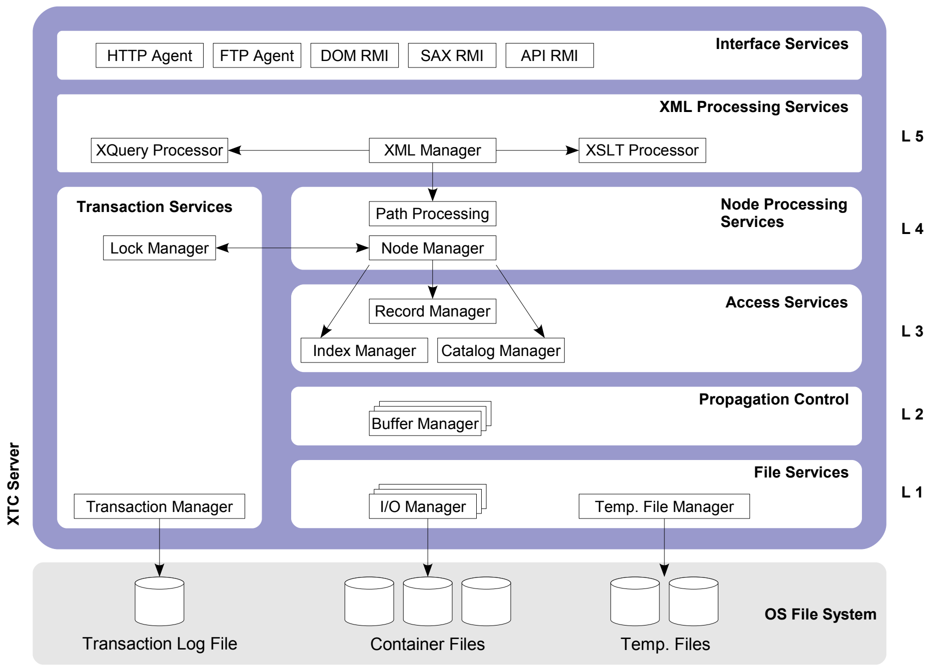


Figure 3.2: XTC Architecture Overview

3.2.2 Distribution

The *XTCDriver* interface is a remote RMI interface. This also allows the XTC JCR implementation itself to be run on distributed remote clients. While the XTC JCR session is designed to run in a single client thread, multiple XTC JCR session instances can access the same repository concurrently. This way the system is designed for scaleability in terms of concurrency which is only limited by the number of concurrent transactions XTC can handle. All JCR internal work (maintaining the transient repository) is done locally and in parallel on the client machines which are synchronized by the XTC database system.

This design enables client applications to directly integrate the XTC JCR implementation as a library. At the same time XTC JCR can be deployed as a resource in a Java EE container (Application Server), letting multiple applications in the container acquire sessions and work with the repository (see deployment diagram 3.3).

3.2.3 XTC JCR Design

Internally XTC JCR is structured in components. Nesting of sub components in large components is common practice in software development. However, it is not easy to “guess” a good component design. In this work an engineering approach is taken to systematically find the sub components in which the system is split up. The component boundaries are cut following the idea of software categories A,R,T,0 or “blood groups” (see chapter four of [Sie04]).

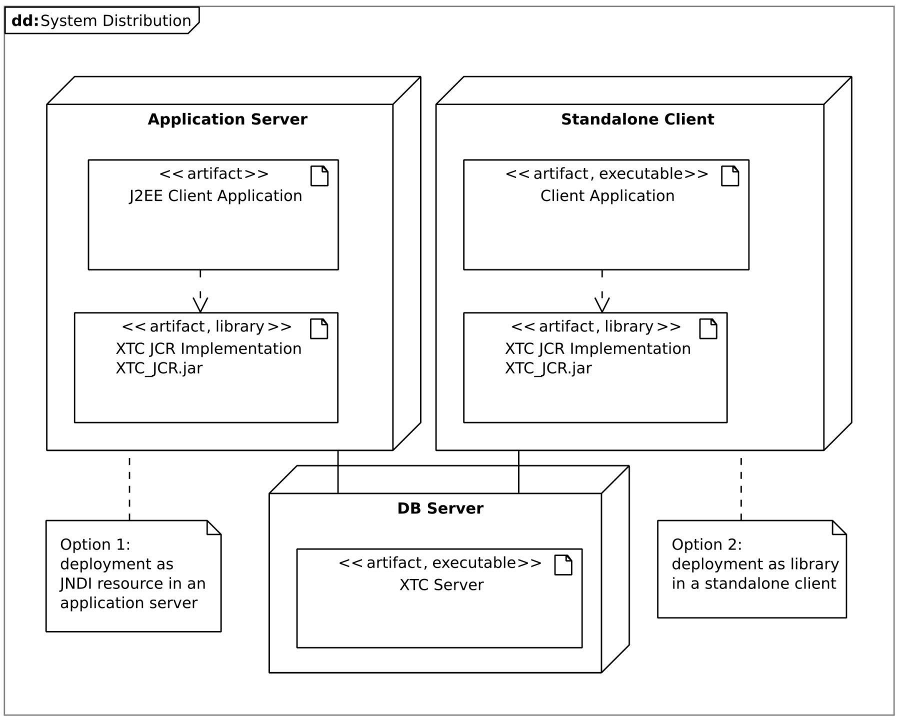


Figure 3.3: XTC JCR Distribution

Software category A stands for "application software", T stands for "technical software", R stands for "transformation software", and 0 is primitive software available for example via the runtime environment. Just like blood groups, software categories cannot be mixed arbitrarily, since the software would clod up to one big unmaintainable chunk otherwise. A good design separates A from T software. This is because A and T change at different rates. A software modification in the database connection code, for example, should not effect a use cases implementation. The software components should use the least common denominator, which is predominantly 0 software, to communicate. This way no unnecessary dependencies arise. In cases where the a application object (e.g. a customer object) needs to be transformed into a technical object (e.g. an entry in a table or XML file) R software comes into play. R software only transforms objects back and forth and connects both worlds.

In this fashion the JSR 170 is implemented in software components of type A (see figure 3.4). There are several A sub components that deal with specific behavior, such as query processing, path processing, transactions, etc. (see table 3.1). This achieves a high coherence within the components and a low coupling between the components. Additionally, these components can be tested independently from one another. All XTC specific behavior is encapsulated in the T typed persistence component. The R type component *JCR2XML* transforms JCR objects into an XML representation and vice versa.

Component	SW-Category	Responsibility
JSR 170 Impl	A	XTC Java Content Repository (JCR) top level implementation: Session, Workspace, Namespace Registry, Transient Repository etc.
Query	A	Creation and management of JCR Queries (XPath).
Path	A	Parsing, validation and transformation of JCR paths.
NodeType	A	Implementation of JCR node type system.
Transaction	A	JTA compliant implementation of a UserTransaction. (Implements optional feature: Transactions)
Value	A	Creation, validation and transformation of JCR Values.
Versioning	A	Workspace versioning. (Implements optional feature: Versioning)
JCR2XML	R	Conversion from JCR content tree to XML tree and vice versa, including query rewriting to fit the internal XML repository format.
JNDI	T	Publishing and configuration of the repository as JNDI resource.
Persistence	T	Decoupling from XTC specific interface. Low-level Transaction and Persistence managers.

Table 3.1: Software Component Overview

3.3 Infrastructure

The infrastructure for this project consists of a T60 Lenovo laptop computer as test and development platform. It runs the XTC Server and a Glassfish application server to demonstrate XTC JCR demo applications in a Java EE container environment.

The application is developed using the well known Netbeans IDE in conjunction with subversion as a version and configuration management system. Development follows a test driven approach using Netbean's integrated JUnit[4] functionality for automated testing.

XTC JCR is a subproject resident in the overall XTC project. In cooperation with the XTC team the Trac system is used to report XTC related bugs and cooperate via the integrated wiki system.

[4] http://www.junit.org/

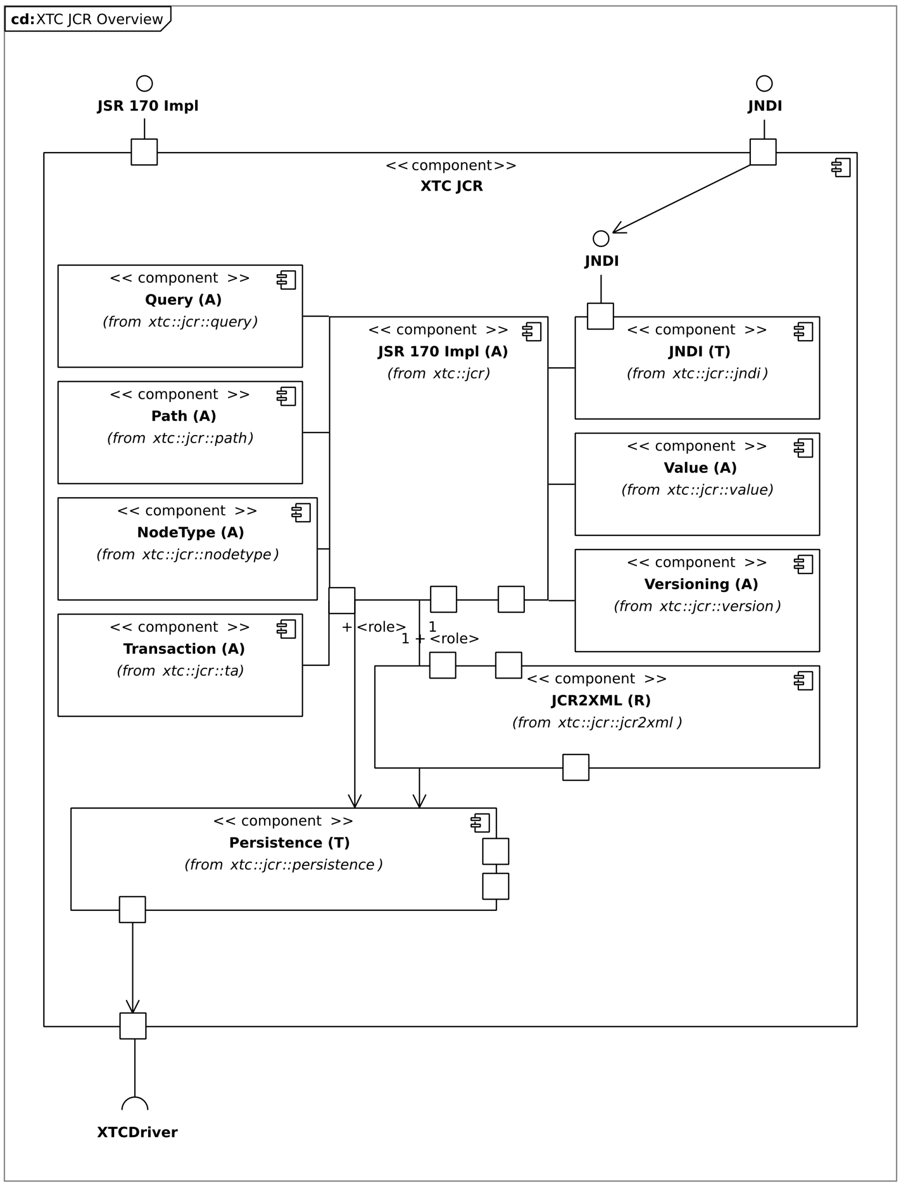


Figure 3.4: XTC JCR Architecture Overview

Chapter 4

Implementation

The implementation is done in the Java programming language which is obvious for a *Java Content Repository.* However, XTC Server and XTC JCR rely on Java 1.6 and are not backwards compatible. The first major difference to the reference implementation. Java 1.6 includes Generics and Enumerations as well as XML related tools (e.g. JAXB) that simply were not present at the time of writing the specification. Therefore, the interfaces do stipulate arrays instead of typed lists and integer constants instead of enumerations. Although the specified interfaces cannot be changed, XTC JCR internally operates these new features and maps the old fashioned interfaces onto the modern implementations. This practice helps to avoid casting and makes the whole application typesafe.

This implementation takes an elegant approach to implement the JSR 170 interfaces (see figure 4.1): For internal processing the types specified by JSR 170 need to be extended with custom methods. These helper methods are part of the interface of the implementing class. However, instead of programming against the implementation of the JSR types, another approach is taken. Between the JSR type and the concrete implementation an additional level of abstraction is introduced in form of an internal interface. By extending the given JCR interface with an internal interface, against which all application code is written, the code stays free of dependencies to the implementing classes. This is useful for unit testing, since this way dummy implementations for the internal interfaces can be written. Furthermore, do these interfaces allow to override the JSR 170 method signatures and provide a more specialized return type.

Listing 4.1 demonstrates this practice by showing how method *addNode* of *javax.jcr.Node* is specialized in *xtc.jcr.NodeInternal* to return the more specific type *NodeInternal.* This practice helps to avoid type casting and encourages reuse of existing functionality.

In order to implement all required parts of the specification a test driven development approach is put in practice. A second source folder exists containing a copy of the package structure in which the test cases for each package are inserted. Prior to implementation black box tests are written for each method of the JSR 170 interface[1]. During the implementation more tests are added, as

[1] These stubs are generated automatically by the IDE.

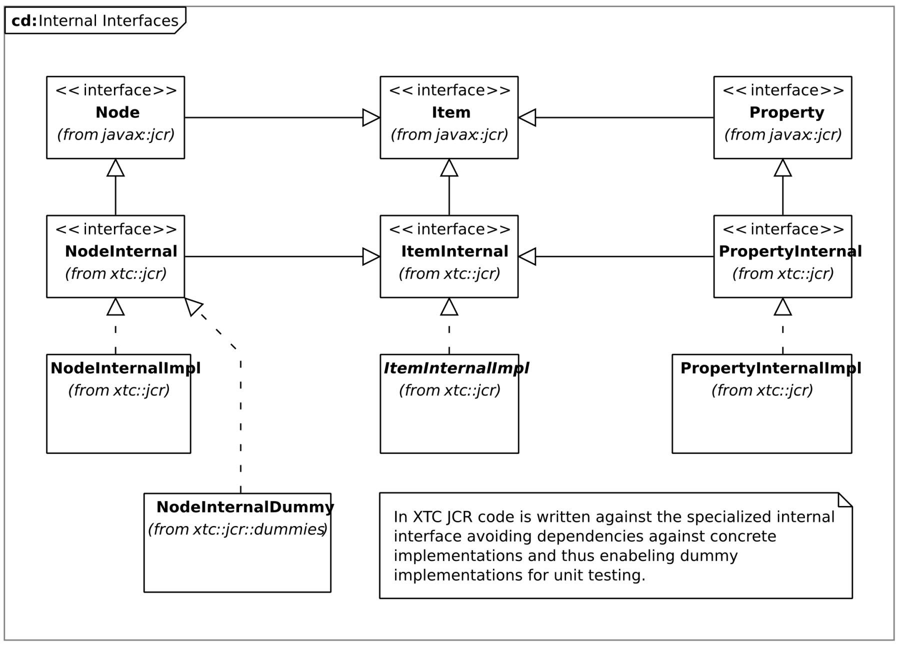


Figure 4.1: Internal interfaces extend JCR interfaces enableing custom interface adaption

required, to cover complicated paths in the program flow. Complete path coverage is not claimed. The test driven development strategy cannot guarantee complete freedom from defects. Inspired by the statistical testing approach the most common cases are reflected in the tests predominantly. Finally, 420 JUnit tests give adequate confidence that the implementation complies to the JSR 170 specification.

```
// JSR 170 Node interface
package javax.jcr;
public interface Node {
  public Node addNode(String relPath) throws ...;
  ...
}

// XTC JCR Node interface
package xtc.jcr;
import javax.jcr.Node;
public interface NodeInternal extends ItemInternal, Node {
  @Override
  public NodeInternal addNode(String relPath) throws ...;
  ...
}
```

Listing 4.1: JSR 170 and XTC JCR *Node* interface

4.1 Level 1

4.1.1 JCR Paths

A JCR workspace is a tree that can be navigated by the client programs. In order to specify the navigation steps JCR paths must be provided[2]. In the JSR 170 interface these paths are provided in form of strings. Eventually, these strings must be parsed and checked for validity before they can be interpreted on top of the tree data. To do so, the *Path* component takes responsibility in managing JCR paths, names, and name patterns.

The specification provides EBNF grammars, defining the correct syntax of paths, names, and name patterns.

```
path                ::= abspath | relpath
abspath             ::= ’/’ relpath | ’/’
relpath             ::= pathelement | relpath ’/’ pathelement
pathelement         ::= name | name ’[’ number ’]’ | ’..’ | ’.’
number              ::= /* An integer > 0 */
name                ::= [prefix ’:’] simplename
simplename          ::= onecharsimplename | twocharsimplename | threeormorecharname
onecharsimplename   ::= /* Any Unicode character except:
                        ’.’, ’/’, ’:’, ’[’, ’]’, ’*’, ’’’, ’"’, ’|’
                        or any whitespace character */
twocharsimplename   ::= ’.’ onecharsimplename | onecharsimplename ’.’
                        | onecharsimplename onecharsimplename
threeormorecharname ::= nonspace string nonspace
prefix              ::= /* Any valid non-empty XML NCName */
string              ::= char | string char
char                ::= nonspace | ’ ’
nonspace            ::= /* Any Unicode character except:
                        ’/’, ’:’, ’[’, ’]’, ’*’,  ’’’, ’"’, ’|’
                        or any whitespace character */
namePattern         ::= disjunct {’|’ disjunct}
disjunct            ::= part [’:’ part]
part                ::= ’*’ | [’*’] fragment {’*’ fragment}[’*’]
fragment            ::= char {char}
```

This small grammar is implemented in hand-written, very efficient parsers. The path parser converts the input string into an abstract syntax tree of the form displayed in figure 4.2.

Within the *Path* component further processing is done in *PathFactory* implementing the commonly known factory pattern [GHJV95]. The abstract syntax tree can be transformed into a *NormalizedPath* or, if applicable, into a *CanonicalPath* instance (see figure 4.7 in chapter 4.1.5). *NormalizedPath* instances are guaranteed to contain only leading *PathParentElements* and no *PathCurrentElements* or trailing slashes. *CanonicalPath* instances are guaranteed to be canonical, i.e. normalized and absolute. More on that in chapter 4.1.5.

4.1.2 Unique Node Identifier

A typical access pattern in a JCR is to acquire the root node and to navigate via the DOM-like direct access methods through the workspace tree. These navigation steps are typically not enclosed by a large transaction. Thus, it is possible that the node acquired gets modified, moved, deleted, or substituted

[2] JCR paths are used in direct access methods and differ from JCR queries (see chapter 4.1.4).

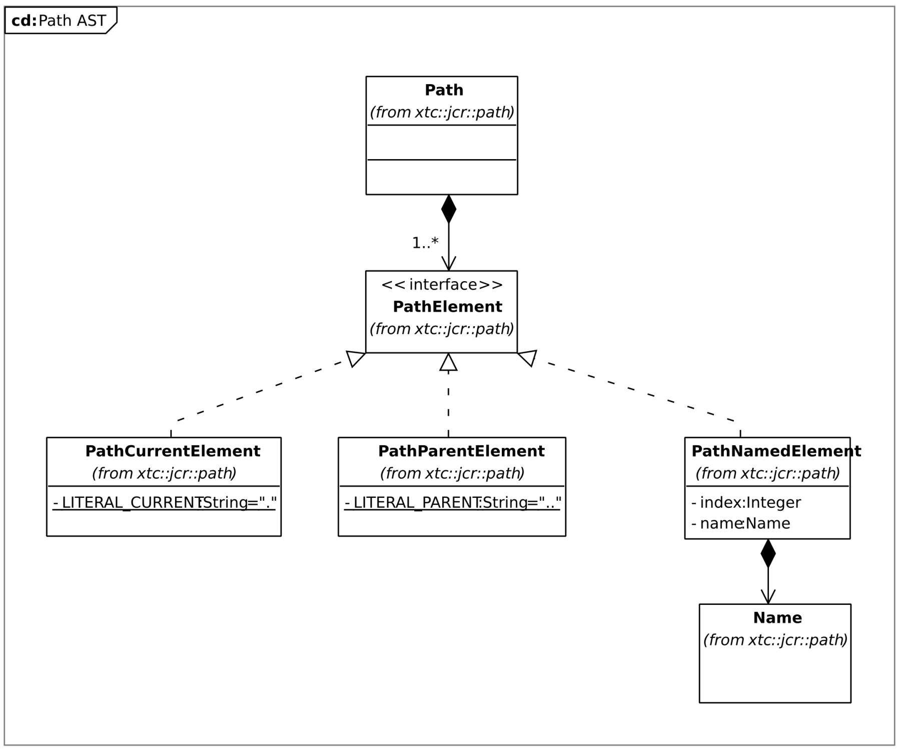


Figure 4.2: Path: Abstract Syntax Tree (AST)

by a concurrent session. In order to correctly handle these situations the implementation requires a mechanism to identify a node unambiguously across transaction boundaries. Only with such an unique node identifier is it possible to find the corresponding node to the local node reference in the database.

The first approach was to use the absolute canonical JCR Path. However, these paths are not stable. As soon as same name siblings are allowed the index of a location step can change. Even worse—the name of a node is not sufficient. Name, index, and node type would need to be stored as unique identifier, since another session could have substituted a node with same name but different type. The check for equality would need to recurse up to the root node and would be very expensive. In order to prevent this, the whole repository would need to be locked which is not an option either. The identity check is a very common operation. Therefore, this approach was discarded.

A second idea was to use XTC's *DeweyIDs* [HH07]. A *DeweyID* is a stable marker for a position in an XML document. Hence, same name siblings would be no problem. In order to test the identity, a *DeweyID* comparison could serve as necessary, yet insufficient, condition—a quick a-priori check. In a second, more expensive step, names and node-types would still need to be checked along the conanical path up to the root node. This approach was discarded for the same reason as the latter one.

As a third approach referenceable nodes and their universal unique identifier

(UUID) property *jcr:uuid* were considered. Checking equality on a UUID is a very quick operation. Additionally, this approach has another advantage. The specification supports moving of nodes, thus changing their position. It turns out that position, as part of a unique identifier, is actually not adequate. This renders the first two approaches even more useless. This approach still has three drawbacks. The first one is obvious. Not every node is referenceable. This could, however, be enforced by the implementation and is legal by the specification. The second is a minor performance issue. The *jcr:uuid* is a JCR property. Depending on the mapping in XML the cost of retrieving the *jcr:uuid* value is almost equal to the cost of retrieving a JCR property. The third drawback is the KO criterion. During moving of a node the implementation must internally rebuild a copy of the moved node. For the time of this operation there exist two logically identical nodes with the same *jcr:uuid* which must not be.

The final solution is an internal XTC JCR specific UUID. In this implementation every node in a workspace has an immutable UUID. The only drawback here is the increased data volume. The advantages are that this concept is powerful enough to handle relocation of nodes and that the check for equality is extremely fast. From a system design standpoint this identifier is a nice solution since it does not mix technical aspects of XTC, such as the *DeweyID*, with JCR application specific aspects, such as a node id. At the same time does XTC accelerate this concept transparently behind the scene. When stored as an XML attribute (see chapter 4.1.3) XTC can leverage the power of an attribute index that, very efficiently, finds the node according to its uuid.

4.1.3 Mapping to XML

The specification distinguishes between two XML views on a workspace, document view and system view. The document view discards some information, such as property arity. It is the format against which JCR XPath queries are virtually run. The system view includes all externally visible information in the repository. It is meant to be an exchange format between repositories.

Neither document nor system view appear to be well suited as storage format. The system view maintains the distinction between node and property as sv:node and sv:property elements. This is really redundant information, since the properties must have a *jcr:type* attribute, and nodes must never have a *jcr:type* attribute. Furthermore, this format is not optimized for XPath query syntax. The document view is problematic when it comes to handling multivalued properties. Multivalued property values are serialized as one string with space as a delimiter. Obviously a regular space needs to be escaped. When importing document view from other sources it is unclear whether the values are escaped multivalue properties or not. More important, this mapping makes answering XPath queries very problematic. XPath and JCR expect a predicate to match a multivalue property when one value of the property matches.

For these reasons this implementation uses an optimized mapping, called "internal view", as internal storage format. The format is a mixture of document and system view. It combines the completeness of the system view and the readable structure of the document view and adds some internal attributes to manage the repository.

Please see the specification for details on how system and document view are mapped. Listing 4.2 shows an example of a workspace mapped to internal view. The internal mapping is constructed as follows[3]:

1. The root of the workspace becomes the XML element *jcr:root*, analogous to the system view.

2. Nodes are mapped to XML elements of the node's name, similar to document view. Each node element carries a *xtcjcr:uuid* attribute with a unique identifier. This unique identifier helps to identify them over transactional boundaries (see chapter 4.1.2). Additionally, a *xtcjcr:definition* attribute carries encoded information on the node's definition, i.e. the declaring node type, the type name, and the required primary types.

3. Properties are mapped to child elements of their parent node element. These elements also carry the name of the property. This name is identical to the attribute name in document view. Yet, the property is represented as an element just like in system view. Additionally, these elements have a *xtcjcr:type* attribute. This attribute differs slightly from the system view's *jcr:type* attribute, that stores the type as string. *xtcjcr:type* stores the corresponding integer constant. Another attribute *xtcjcr:multiValued* carries a boolean value. This value is *true* in case the property is a multivalued property. The value of a property is mapped to a *sv:value* child element with its value as text node, just like in system view. In case of a multivalued property the property can contain multiple such *sv:value* child elements in order of the value array returned by *Property.getValues()*.

4.1.4 Declarative Queries

The JCR XPath query language is a subset of XPath 2.0 [BBC$^+$07]. The specification supplies a grammar that specifies the JCR query language which is virtually interpreted on the *document view* XML representation of the *Content Repository*. In order to process this language and run queries on XTC's internal XML representation of the workspace a query rewriting engine is required that transforms the original query into a query on the internal format.

The query rewriting engine consists of a parser and rewriting stages. The parser generates an abstract syntax tree. The rewriting stages transform this tree into a tree that represents an XQuery statement, applicable to the underlying workspace document in XTC.

The parser and the accompanying lexer were generated from the supplied grammar using JavaCC[4]. In order to transform the abstract syntax tree a generic transformation framework was implemented. Each rewriting stage (see figure 4.4) gets registered in the rewriter. For each query the rewriter applies all stages in sequence of their registration. Each rewriting stages takes the abstract syntax tree of the prior stage as input and returns the transformed syntax tree.

[3] The namespace *xtcjcr* is reservered to XTC JCR specific names.

[4] `https://javacc.dev.java.net`

```
<jcr:root xtcjcr:uuid="a6af9bd9-9c5b-4b5e-9d87-d47d8ea65ade"
     xtcjcr:definition="xtcjcr:rootDeclaringType/xtcjcr:root/[
     xtcjcr:root]">
  <jcr:primaryType xtcjcr:type="7" xtcjcr:multiValued="false">
    <sv:value>
      xtcjcr:root
    </sv:value>
  </jcr:primaryType>
  <jcr:mixinTypes xtcjcr:type="7" xtcjcr:multiValued="true" />
  <node0 xtcjcr:uuid="804b8272-cec2-44b7-b949-c47a4a3375e8"
       xtcjcr:definition="xtcjcr:root/:*/[nt:base]">
    <jcr:mixinTypes xtcjcr:type="7" xtcjcr:multiValued="true">
      <sv:value>
        mix:referenceable
      </sv:value>
    </jcr:mixinTypes>
    <jcr:primaryType xtcjcr:type="7" xtcjcr:multiValued="false">
      <sv:value>
        nt:unstructured
      </sv:value>
    </jcr:primaryType>
    <jcr:uuid xtcjcr:type="1" xtcjcr:multiValued="false">
      <sv:value>
        c0b12360-8b0a-4cf7-9505-e67a05c3898f
      </sv:value>
    </jcr:uuid>
  </node0>
</jcr:root>
```

Listing 4.2: XML mapped workspace with one referenceable node named *node0*

The rewriting stages are realized following the *Visitor* pattern [GHJV95] (see figure 4.3). The Visitor pattern enables supplementary addition of operations to classes which implementation cannot be changed. It is convenient to leave the implementation of the abstract syntax tree classes untouched, since they are generated via JavaCC. Instead of distributing the implementation of one rewriting stage into several classes of the abstract syntax tree, the implementation fits coherently into a single rewriting stage class. The code for each rewriting rule is nicely separated from all other rules.

The rewriter executes each stage as shown in figure 4.6. In the abstract syntax tree all references to child nodes are of type *MySimpleNode*. In order to determine the visitor's *visit*[5] method according to the real parameter type a double-dispatch mechanism is used. As a default behavior for all visit methods a depth-first traversal of the syntax tree is implemented via recursive descent in the *AbstractXPathAdapterVisitor*.

[5] See the figure 4.5 for an overview of all visitor methods.

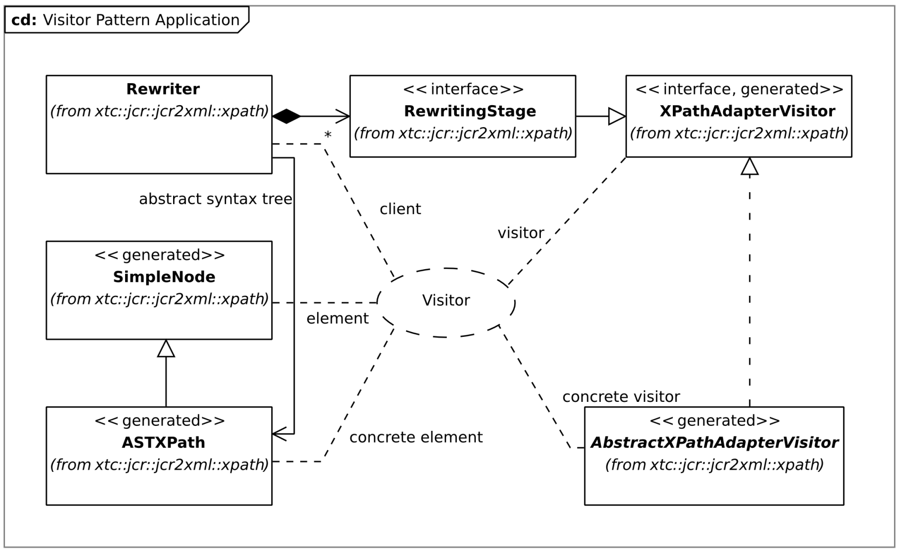

cd: Visitor Pattern Application
Rewriter
(from xtc::jcr::jcr2xml::xpath)
<< interface>>
RewritingStage
(from xtc::jcr::jcr2xml::xpath)
<< interface, generated>>
XPathAdapterVisitor
(from xtc::jcr::jcr2xml::xpath)
*
abstract syntax tree
client
visitor
<< generated>>
SimpleNode
(from xtc::jcr::jcr2xml::xpath)
Visitor
element
concrete visitor
<< generated>>
ASTXPath
(from xtc::jcr::jcr2xml::xpath)
concrete element
<< generated>>
AbstractXPathAdapterVisitor
(from xtc::jcr::jcr2xml::xpath)

Figure 4.3: Visitor pattern applied on JavaCC generated classes

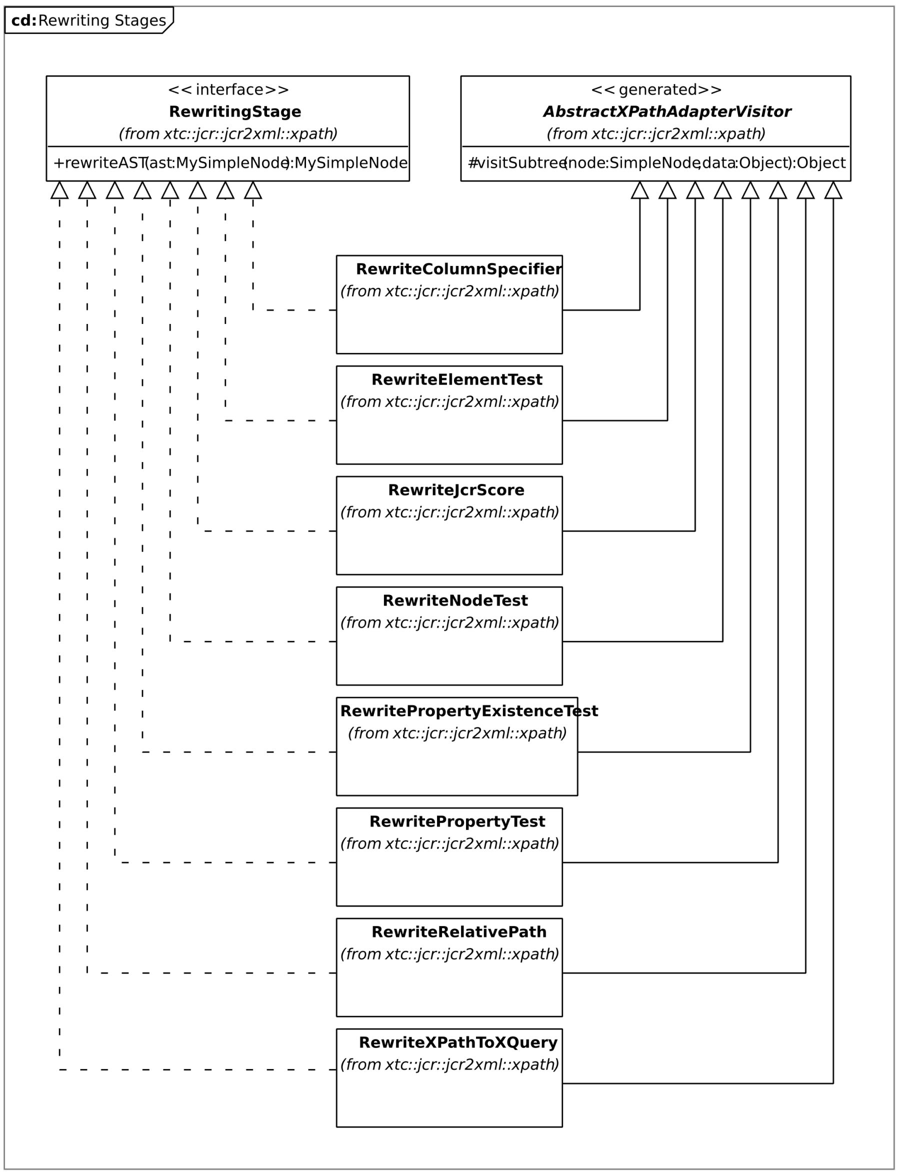

cd: Rewriting Stages
<< interface>>
RewritingStage
(from xtc::jcr::jcr2xml::xpath)
+rewriteAST(ast:MySimpleNode):MySimpleNode
<< generated>>
AbstractXPathAdapterVisitor
(from xtc::jcr::jcr2xml::xpath)
#visitSubtree(node:SimpleNode,data:Object):Object
RewriteColumnSpecifier
(from xtc::jcr::jcr2xml::xpath)
RewriteElementTest
(from xtc::jcr::jcr2xml::xpath)
RewriteJcrScore
(from xtc::jcr::jcr2xml::xpath)
RewriteNodeTest
(from xtc::jcr::jcr2xml::xpath)
RewritePropertyExistenceTest
(from xtc::jcr::jcr2xml::xpath)
RewritePropertyTest
(from xtc::jcr::jcr2xml::xpath)
RewriteRelativePath
(from xtc::jcr::jcr2xml::xpath)
RewriteXPathToXQuery
(from xtc::jcr::jcr2xml::xpath)

Figure 4.4: Concrete Rewriting Stages

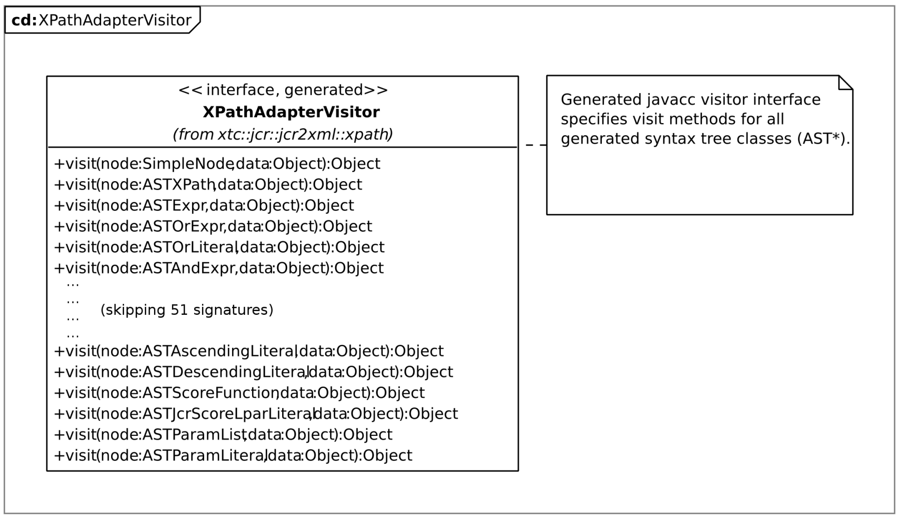

cd:XPathAdapterVisitor
<< interface, generated>>
XPathAdapterVisitor
(from xtc::jcr::jcr2xml::xpath)
+visit(node:SimpleNode,data:Object):Object
+visit(node:ASTXPath,data:Object):Object
+visit(node:ASTExpr,data:Object):Object
+visit(node:ASTOrExpr,data:Object):Object
+visit(node:ASTOrLiteral,data:Object):Object
+visit(node:ASTAndExpr,data:Object):Object
...
(skipping 51 signatures)
+visit(node:ASTAscendingLiteral,data:Object):Object
+visit(node:ASTDescendingLiteral,data:Object):Object
+visit(node:ASTScoreFunction,data:Object):Object
+visit(node:ASTJcrScoreLparLiteral,data:Object):Object
+visit(node:ASTParamList,data:Object):Object
+visit(node:ASTParamLiteral,data:Object):Object
Generated javacc visitor interface specifies visit methods for all generated syntax tree classes (AST*).

Figure 4.5: Generated visitor interface with signatures for all abstract syntax tree classes.

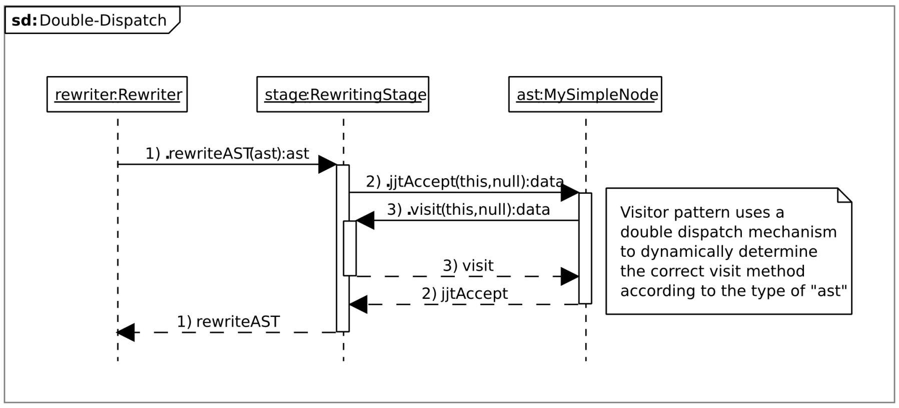

sd:Double-Dispatch
rewriter:Rewriter
stage:RewritingStage
ast:MySimpleNode
1) .rewriteAST(ast):ast
2) .jjtAccept(this,null):data
3) .visit(this,null):data
3) visit
2) jjtAccept
1) rewriteAST
Visitor pattern uses a double dispatch mechanism to dynamically determine the correct visit method according to the type of "ast"

Figure 4.6: Double dispatch mechanism in visitor pattern implementation.

The rewriting stages in order of application are:

RewriteColumnSpecifier JCR queries support column specifiers, selecting the properties, to be returned in the table-like query result. This rewrite rule determines all column specifiers, for later use in the result, and removes them from the query.
Example: *//cb:articles/@title*
into *//cb:articles*

RewriteRelativePath Relative queries are always interpreted relative to the repository's root node. Therefore, all relative paths are transformed into absolute paths.
Example: *cb:pages/cb:articles/*[@author = "sweety84"]*
into: */jcr:root/cb:pages/cb:articles/*[@author = "sweety84"]*

RewriteNodeTest In order to ensure that the resulting XML elements are all JCR nodes, and not elements representing a JCR property, a node test predicate is appended to all node steps. The predicate test consists of a test for the *jcrxtc:uuid* attribute which every node element must carry.
Example: *//cb:articles/cb:pages*
into *//cb:articles[@jcrxtc:uuid]/cb:pages[@xtcjcr:uuid]*

RewriteElementTest The type test in form of *element(name,type)* is rewritten into a predicate on node step *name*. The predicate consists of a list of "or" concatenated string comparisons of *jcr:primaryType* child against all subtypes of *type*.
Example: *//element(abc,nt:base)*
into *//abc[jcr:primaryType = "nt:base" or jcr:primaryType="..."]*

RewritePropertyExistenceTest JCR properties are internally stored as XML child elements. Each such element has a *xtcjcr:type* property storing the property type as an integer. When a predicate tests property existence the predicate has to be rewritten from *[@$propertyName$]* into *[$propertyName$/@xtcjcr:type]*[6].

RewritePropertyTest JCR properties are internally stored as XML child elements. Each element representing a JCR property contains a list of *sv:value* child elements. When a property value is compared to a literal in a predicate, this rewriting rule transforms *@$propertyName$* into *$propertyName$/sv:value*. For multi valued properties the semantic of such a comparison is: at least one value in the list must match. The transformation preserves this semantic.

RewriteXPathToXQuery The XPath query and the *order by* clause are translated into an XQuery FLWOR[7] expression.

[6] Each child element contains itself a list of *sv:value* child elements. The "sv:value" child element must not be used to test property existence, since multi valued properties may exist with a list of empty values.

[7] FOR LET WHERE ORDERBY RETURN (FLWOR)

RewriteJcrScore The *order by* clause in a JCR query supports a function called *jcr:score()*. The semantics of this function is left to the implementation. Since XTC does not support a fuzzy matching, a match score cannot be computed. Thus, this function must be ignored in the query. Example: *//cb:articles order by jcr:score() ascending* into *//cb:articles order by ascending*

The specification provides EBNF grammars, defining the correct syntax of JCR queries.

```
ExprComment        ::= "(:" (ExprCommentContent | ExprComment)* ":)"
ExprCommentContent ::= Char
IntegerLiteral     ::= Digits
DecimalLiteral     ::= ("." Digits) | (Digits "." [0-9]*)
DoubleLiteral      ::= (("." Digits) | (Digits ("." [0-9]*)?)) ("e" | "E")
                       ("+" | "-")? Digits
StringLiteral      ::= ('"' (('"' '"') | [^"])* '"') | ("'" (("'" "'") | [^'])* "'")
Digits             ::= [0-9]+
NCName             ::= [http://www.w3.org/TR/REC-xml-names/#NT-NCName]
QName              ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]
Char               ::= [http://www.w3.org/TR/REC-xml#NT-Char]
XPath              ::= Expr?
Expr               ::= ExprSingle
ExprSingle         ::= OrExpr
OrExpr             ::= AndExpr ( "or" AndExpr )*
AndExpr            ::= InstanceofExpr ( "and" InstanceofExpr )*
InstanceofExpr     ::= TreatExpr
TreatExpr          ::= CastableExpr
CastableExpr       ::= CastExpr
CastExpr           ::= ComparisonExpr
ComparisonExpr     ::= RangeExpr (( GeneralComp | GeneralComp) RangeExpr )?
RangeExpr          ::= AdditiveExpr
AdditiveExpr       ::= MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
MultiplicativeExpr ::= UnaryExpr
UnaryExpr          ::= ("-" | "+")* UnionExpr
UnionExpr          ::= IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
/* Note that support for a UnionExpr of attributes in the
last location step is optional*/

IntersectExceptExpr ::= ValueExpr
ValueExpr          ::= PathExpr
PathExpr           ::= ("/" RelativePathExpr?)
                       | ("//" RelativePathExpr)
                       | RelativePathExpr
RelativePathExpr   ::= StepExpr (("/" | "//") StepExpr)*
StepExpr           ::= AxisStep | FilterStep
AxisStep           ::= (ForwardStep) Predicates
FilterStep         ::= PrimaryExpr Predicates
ContextItemExpr    ::= "."
PrimaryExpr        ::= Literal | VarRef | ParenthesizedExpr |
                       ContextItemExpr | FunctionCall
Predicates         ::= ("[" Expr "]")*
GeneralComp        ::= "=" | "!=" | "<" | "<=" | ">" | ">="
ForwardStep        ::= AbbrevForwardStep
AbbrevForwardStep  ::= "@"? NodeTest
NodeTest           ::= KindTest | NameTest
NameTest           ::= QName
Wildcard           ::= "*" | <NCName ":" "*"> | <"*" ":" NCName>
Literal            ::= NumericLiteral | StringLiteral
NumericLiteral     ::= IntegerLiteral | DecimalLiteral |
                      DoubleLiteral
ParenthesizedExpr  ::= "(" Expr? ")"
FunctionCall       ::= <QName "("> (ExprSingle ("," ExprSingle)*)? ")"
KindTest           ::=  ElementTest
ElementTest        ::= <"element" "(">
                      | (ElementNameOrWildcard (","
                      TypeNameOrWildcard? ) )? ")"
ElementName        ::= QName
AttributeName      ::= QName
TypeName           ::= QName
```

```
ElementNameOrWildcard ::= ElementName | "*"
TypeNameOrWildcard ::= TypeName | "*"
JCRXPathExpr       ::= (XPath OrderByClause?)?
OrderByClause      ::= "order by" OrderSpecList
OrderSpecList      ::= OrderSpec ("," OrderSpec)*
OrderSpec          ::= ("@" AttributeName OrderModifier) |
                       (ScoreFunction OrderModifer)
OrderModifier      ::= ("ascending" | "descending")?
ScoreFunction      ::= "jcr:score(" ParamList ")"
ParamList          ::= /* 0..* comma separated parameters */
```

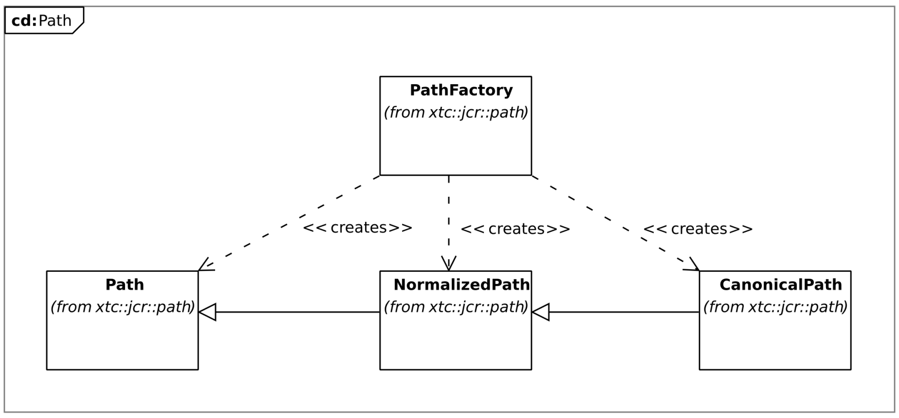


Figure 4.7: Path system: PathFactory and assertion types.

4.1.5 Type-level Assertion

In general, public methods that can be called from an unknown context, such as a JCR client, must perform a number of validation steps each time they are called. Typically this includes a sanity check of the current context and input parameter validation against the specification. For methods that are called from within the implementation only, the context is known and the methods can be designed by contract, omitting these checks. This chapter presents an elegant approach that uses Java's type system to explicitly enforce assertions on the input parameters statically, without runtime overhead, in a design by contract situation. This idea is inspired by the "Trusted Kernel" approach demonstrated on functional programming languages [KS07].

Such a situation arises for example when a canonical JCR Path value is expected as an input value. A naive implementation could design a method to accept a String value as a parameter. The type String does not even ensure that the character sequence is a valid path, nor does it ensure that the path is canonical. A better approach could be to encapsulate a parsed path in a class and equip the class with boolean properties: *isNormalized()*, *isCanonical()*, etc. However, these properties would still need to be checked at several places at runtime. Instead, the requirement on the input parameter can be made explicit by expecting a value of a specialized type: CanonicalPath (see figure 4.7).

Concrete path instances are created using an implementation of the factory pattern [GHJV95]. All path parsing and assertion checking is done in the factory and within the Path component. Finally, *NormalizedPath* instances are guaranteed to have only leading .. and no . or trailing /. *CanonicalPath* instances are guaranteed to be canonical, i.e. normalized and absolute. Another elegant effect of using these types (see figure 4.7) is that part of the post condition of a method can be manifested in the returned type. For example the following method signature does guarantee that the returned path is canonical.

The same approach is followed for the implementation of JCR names and name patterns.

```
public CanonicalPath createCommonAncestorPath(
    CanonicalPath cSourcePath, CanonicalPath cDestPath);
```

Listing 4.3: Post condition manifested in return type.

4.2 Level 2

The implementation of a level 2 repository focuses on the development of a namespace registry and operations on the transient repository layer. In general, read write functionality also imposes the need for transaction management. These topics are dealt with in this chapter.

Figure 4.8 shows the design and interconnections of the XTC JCR components. The transactional behavior is controlled via the TransactionManager interface from within the application components in *JCR 170 Impl.* The PersistenceManager interface is operated from the *JCR2XML* layer.

4.2.1 Namespace Registry

A JCR contains a namespace system modeled analogously to XML's namespaces. The namespace registry maps shorthand prefixes to namespaces. Namespaces are URIs. All JCR names may have a prefix, delimited by a single colon character indicating their namespace. This way naming collisions can be minimized. This fact is exploited by the specification and by this implementation in that certain namespaces are reserved to JCR and XTC JCR. All reserved namespaces are listed in table 4.1.

Prefix	URI	Namespace Description
jcr	http://www.jcp.org/jcr/1.0	namespace for items defined by built-in node types
nt	http://www.jcp.org/jcr/nt/1.0	namespace for built-in primary node types
mix	http://www.jcp.org/jcr/mix/1.0	namespace for built-in mixin node types
xml	http://www.w3.org/XML/1998/namespace	namespaces that must not be redefined and should not be used (read http://www.w3.org/TR/REC-xml-names/#ns-qualnames)
	(empty prefix and uri)	default namespace (empty uri)
xmlns	http://www.w3.org/2000/xmlns	prefix to declare namespaces
sv	http://www.jcp.org/jcr/sv/1.0	namespace used in the system view XML serialization format
xsd	http://www.w3.org/2001/XMLSchema	XML schema namespace
xsi	http://www.w3.org/2001/XMLSchema-instance	XML schema instance namespace
xtcjcr	http://wwwlgis.informatik.uni-kl.de/jcr/xtcjcr/1.0	namespace used in the internal view XML serialization format

Table 4.1: Reserved namespaces in XTC JCR

Neither the prefix nor the URI of a reserved namespace may be redefined. Every *Level 1* repository must contain a namespace registry and support jcr, nt, mix, xml and the empty prefix. A *Level 2* repository must futhermore support the registration of custom namespaces.

The namespace registry is global to all workspaces in a repository. XTC does not provide special features to handle namespaces. Hence, this feature was implemented in a straight forward fashion. All prefix to URI mappings are maintained in an XML file named namespaceRegistry.xml on the XTC server. All sessions query this central resource and resolve prefixes to URIs and vice versa. However, as an optimization, the reserved mappings are kept as constants in the code and custom namespace mappings are cached for the time of a transaction to avoid unnecessary and expensive network traffic.

Listing 4.4 shows two imaginary custom namespaces.

```
<?xml version="1.0"?>
<registry>
  <namespace prefix="abc" uri="http://example.com/abc"/>
  <namespace prefix="def" uri="http://example.com/def"/>
</registry>
```

Listing 4.4: namespaceRegistry.xml

It is not supported to remove a URI from the namespace registry, since it cannot be made sure that this URI is not in use in another session. For example *unregisterNamespace("abc")* would result in an exception. This behavior corresponds to the behavior of the reference implementation. For the same reason it is not possible to remap an existing prefix to a new URI, since this would incorporate removing the old URI from the registry.

It is still possible to remap another prefix to an existing URI. For example *registerNamespace("hij", "http://example.com/abc")* would succeed and result in the mapping shown in listing 4.5.

```
<?xml version="1.0"?>
<registry>
  <namespace prefix="hij" uri="http://example.com/abc"/>
  <namespace prefix="def" uri="http://example.com/def" />
</registry>
```

Listing 4.5: namespaceRegistry.xml after *registerNamespace* operation

4.2.2 Transient Repository

The specification stipulates a transient and a persistent layer. Changes in a session are transient, meaning invisible to other sessions. They stay visible to the current session only up to the point in time when they are saved via *Item.save()* or *Session.save()*. A save on an item persists all the changes on this item and the underlying subtree. A save on the session persists all changes in the workspace and is equivalent to a save on the root node (see chapter 2.4).

As a theoretical basis a state machine for each item's state was developed. Figure 4.9 shows how an item can have a representation in either the transient repository or in the persistent repository. There are also situations where an item can have a persistent and a transient representation simultaneously.

During all operations, such as retrieval of child items, retrieval of item paths, or addition, removal, or modification of items, all transient changes in the current session must be respected as if they were already applied to the repository. This renders attempts to map retrieval operations directly to XTC impossible, since transient and persistent items need to be merged. For example retrieval of a node's JCR path can only be answered from within the session. Nodes along the path could be transient or same name sibling nodes could have been inserted or removed in the transient repository increasing or decreasing indexes along the path.

The implementation is designed to manage transient representations of items in the $TransientRepository$ instance of a session. This object contains two Java maps: $transientNodes$ and $transientProperties$. These maps can be thought of as partial functions.

$$\begin{aligned} &tansientNodes: && NodeId \rightarrow NodeInternal \\ &transientProperties: && NodeId \times Name \rightarrow PropertyInternal \end{aligned}$$

As invariants for $transientNodes$ and $transientProperties$ the implementation guarantees:

$$\begin{aligned} &\forall n \in NodeInternal: \\ &n.itemState \in \{CHANGED, NEW, DESTROYED\} \\ &\Leftrightarrow n \in transientNodes(NodeId) \end{aligned}$$

$$\begin{aligned} &\forall p \in PropertyInternal: \\ &p.itemState \in \{CHANGED, NEW, DESTROYED\} \\ &\Leftrightarrow p \in transientProperties(NodeId, Name) \end{aligned}$$

This means that all nodes referenced in the map are valid transient nodes and all transient nodes are unexceptionally registered in the map. The same holds for JCR properties. Thanks to the bijective mapping between $NodeId$ and $NodeInternal$ it is simple and efficient to find out whether a persistent node with a certain $nId \in NodeId$ has a transient representation. It only has to be tested whether there exists a mapping so that $transientNodes(nId)$ is defined. If this is the case the transient version of the node shadowing the persistent node can directly be retrieved from the map: $transientNodes(nId)$.

For the implementation of $transientNodes$ a *java.util.LinkedHashMap* was chosen. This implementation of the *Map* interface is even more powerful than the notion of a function as it is able to preserve insertion order. This is necessary to replay all transient operations in correct order on a subtree *save*. The use of *HashMap* requires an efficient implementation of the *NodeId.hashCode()* method. For that reason does every *NodeId* instance precompute the hash code during construction and merely return that constant value on *hashCode()* invocation.

Node.addNode

New nodes can only be created by adding them as new child nodes under an existing (transient or persistent) node. The root node does always exist. The implementation ensures that the root node cannot be deleted.

```java
// JSR 170 Node interface
package javax.jcr;
public interface Node {
  public Node addNode(String relPath) throws ...;
  public Node addNode(String relPath, String
      primaryNodeTypeName) throws ...
  ...
}
```

Listing 4.6: JSR 170 *addNode* methods

The JCR node interface contains two *addNode* signatures (see listing 4.6). The *relPath* parameter in both methods can actually be a relative path to the parent node of the node to be added, appended with the name desired for the new node. An index at the last step is not allowed. New nodes are always appended last to the list of child nodes.

The first method is actually equivalent to calling the second method with *null* as second parameter. The implementation maps each method call in this fashion to the second method signature.

In case that the *primaryNodeTypeName* is *null* the implementation tries to automatically determine the appropriate node type by looking at the parent's definition and the name of the new node. In cases where a *primaryNodeTypeName* is provided it is checked whether this nodetype is allowed to be applied.

Once a new *Node* instance is created it is registered in the *transientNodes* map with item state NEW.

In order to manage the parent child relation for NEW nodes, each node maintains a list of all NEW child nodes. New child nodes are added to that list in creation order. Since this list must be modified on an *addNode* operation, the parent node is added to *transientNodes* having item state CHANGED.

Node.setProperty

```java
// JSR 170 Node interface
package javax.jcr;
public interface Node {
  setProperty(String name, Value value, int type) throws
      ...
  ...
}
```

Listing 4.7: JSR 170 *setProperty* methods

JCR properties are designed to be typed. The JCR node interface contains 14 setProperty signatures. They provide convenient ways of setting property values of different property types. If a value of a different type is provided a best-effort conversion is conducted or an exception is thrown. The basic procedure, which is common to all setProperty methods, starts with retrieving the property, if it exists, or creating a new property of the specified type and name. In a second step the application of the new value is delegated to the *Property.setValue* method described in chapter 4.2.2.

If the parent node of the property is not already a transient node, it is registered in the *transientNodes* under item state CHANGED. If the property did not exist before, it gets registered in *transientProperties* under the item state NEW.

Property.setValue

After setting the value of property the implementation makes sure the property is transient. In case of a *null* value the property gets marked as DESTROYED. In the other case the property stays in state NEW or turns from state PERSISTENT to CHANGED. Finally, the implementation ensures that the parent node of the property is transient and that the invariants for *transientNodes* and *transientProperties* hold.

Item.remove

The effect of removing an item depends on whether the item has a persistent representation or not. In cases where the item only has a transient representation, when item state equals NEW, all traces of that item are removed from the transient repository, and the item is treated as if had never existed. The item gets marked as INVALIDATED in order to inhibit any further operation on that instance.

Items that do have a persistent representation are set to item state DESTROYED. When removing a node it is made sure that this destroyed node is registered in *transientNodes*. When removing a property that has a persistent representation it is made sure that the parent node, which must have a persistent representation as well, is registered in *transientNodes* in item state CHANGED. The property itself is registered in *transientProperties*.

Node.orderBefore

The specification allows reordering of child nodes. The method *Node.orderBefore* allows to place a child node in front of another child node (called *orderBefore move*). When a *null* value is provided instead of a destination name the child node is moved to the end of the child node list.

The *transientNodes* map preserves insertion order, but it cannot be used to manage child node order.

OrderBefore moves are not commutative. Depending on the order of how the *orderBefore moves* are applied different permutations can be produced. Each node therefore keeps a history of *orderBefore moves* on its children. As soon as a *orderBefore* operation is applied the parent node changes into a transient state. Whenever child nodes of such a node are retrieved and on *save* the *orderBefore history* is replayed and the child node list is sorted according to these moves.

The *addNode* operation always appends new nodes last. All *orderBefore* entries in the history prior to that addition stay intact, but all histroy entries that move a child node to the end of the list must be rewritten to move the node in front of the new node.

The history must be rewritten as well when a child node gets removed. In this case all entries having the removed node as a destination node must be redirected to the next child node in the list.

Other transient operations

All other transient operations can be performed by delegation to the methods above. These methods are:

- *Node*: *addMixin*, *removeMixin* (see chapter 4.2.3)
- *Session*: *move* , *importXML* (see chapter 4.2.4)
- *Query*: *storeAsNode*

Session.save & Item.save

Save on a session is equivalent to calling *save* on the root node. In general *save* can be called on any item. This item must not be of state INVALIDATED, NEW, or DESTROYED or an exception will be thrown. Thus, states CHANGED and PERSISTENT are allowed. If the item is a property the saving of the property's value(s) is performed and the property's transient state reverts to PERSISTENT. The other case (the item is a node) is much more complicated. In this case all changes in the underlying subtree must be persisted. The implementations does this in the following steps:

1. When a node gets moved it is internally represented as removing the node at its old position and inserting a copy of that node in the new position. A *save* on either of these two internal nodes must include the fellow node. Thus, the *save* must be called on a common ancestor. This implementation calls this the *fellowNode constraint*. In this first step the program finds all transient nodes affected by this *save* and checks the *fellowNodes constraint*.
2. *transientNodes* and *transientProperties* are cleaned from all transient nodes and properties that belong to subtrees of DESTROYED nodes affected by this *save*. While doing so a referential integrity check is performed. This is an expensive operation that finds all referenceable nodes in the subtree of each DESTROYED node.
3. All transient nodes are traversed in insertion order and changes are persisted. It is noteworthy that there exist different methods of inserting new nodes. New subtrees are inserted as a whole in form of serialized XML. Changes are applied via DOM methods.

4. All *orderBefore histories* of nodes of node state CHANGED are replayed and the order of child nodes is changed in XTC. The *orderBefore history* of NEW nodes must not be replayed here since it was already respected in the last step when the child nodes were inserted as part of the subtree serialization.

5. In the last step all transient nodes and properties affected by this save operation are removed form the transient maps.

4.2.3 Node Types

The JSR 170 specification defines a node type system in chapter 6.7 that must support single inheritance. Please see the specification for a detailed explanation on the type system.

The supported standard node types are configured in an XML file[8] taken from the reference implementation. In order to process this file an XML Schema[9] was reverse engineered that describes the structure of the node type configuration file. This implementation takes advantage of *Java Architecture for XML Binding* (JAXB)[10] technology, specified in JSR 222 [Kaw06]. JAXB is part of Java SE version 1.6. Using JAXB Java classes were generated using the schema file. These classes automatically unmarshal the node definitions in XML. Using some "glue code" the generated bindings were used to implement the required JCR type system.

```
// JSR 170 Node interface
package javax.jcr;
public interface Node {
 public void addMixin(String mixinName) throws ...
 public void removeMixin(String mixinName) throws ...
   ...
}
```

Listing 4.8: JSR 170 mixin methods

Mixins are types that can be added and removed after creation. A node can have multiple mixin types. *Node.addMixin* and *Node.removeMixin* (see listing 4.8) are internally mapped to the multivalued property *jcr:mixinTypes*. So genreally no special treatment is required. However, there exists hardcoded behavior for mixin type *mix:referenceable*, since here UUIDs must be genereated automatically.

4.2.4 Import

A *Level 2* repository must support import of serialized content in form of plain XML or SAX Parser events. Internally plain XML input is parsed so the imple-

[8] see file *builtin_nodetypes.xml*

[9] see file *nodetypes.xsd*

[10] `https://jaxb.dev.java.net/`

mentation only has to deal with the generated SAX Parser events as input. Two formats are supported: System View and Document View import. The implementation automatically determines the appropriate import mechanism. The software is implemented in a fashion utilizing the *Strategy Pattern* [GHJV95], i.e. once the import format is detected all actions are delegated to the appropriate strategy (*systemview* or *documentview* importer).

The *systemview* importer is a state machine, reacting on events during parsing of the imported document. The state graph is illustrated in figure 4.10. This automat is implemented using the *State Pattern* [GHJV95]. The reason why there are six states instead of only three is that a node can only be created once it is known where it is supposed to be created at. In the specification several behaviors of how to handle duplicated UUIDs are specified. To handle these behaviors the UUID of a referenceable node must be known prior to its creation. The state machine needs to “wait” for the *jcr:uuid* property which comes at third position in each serialized node in system view import format.

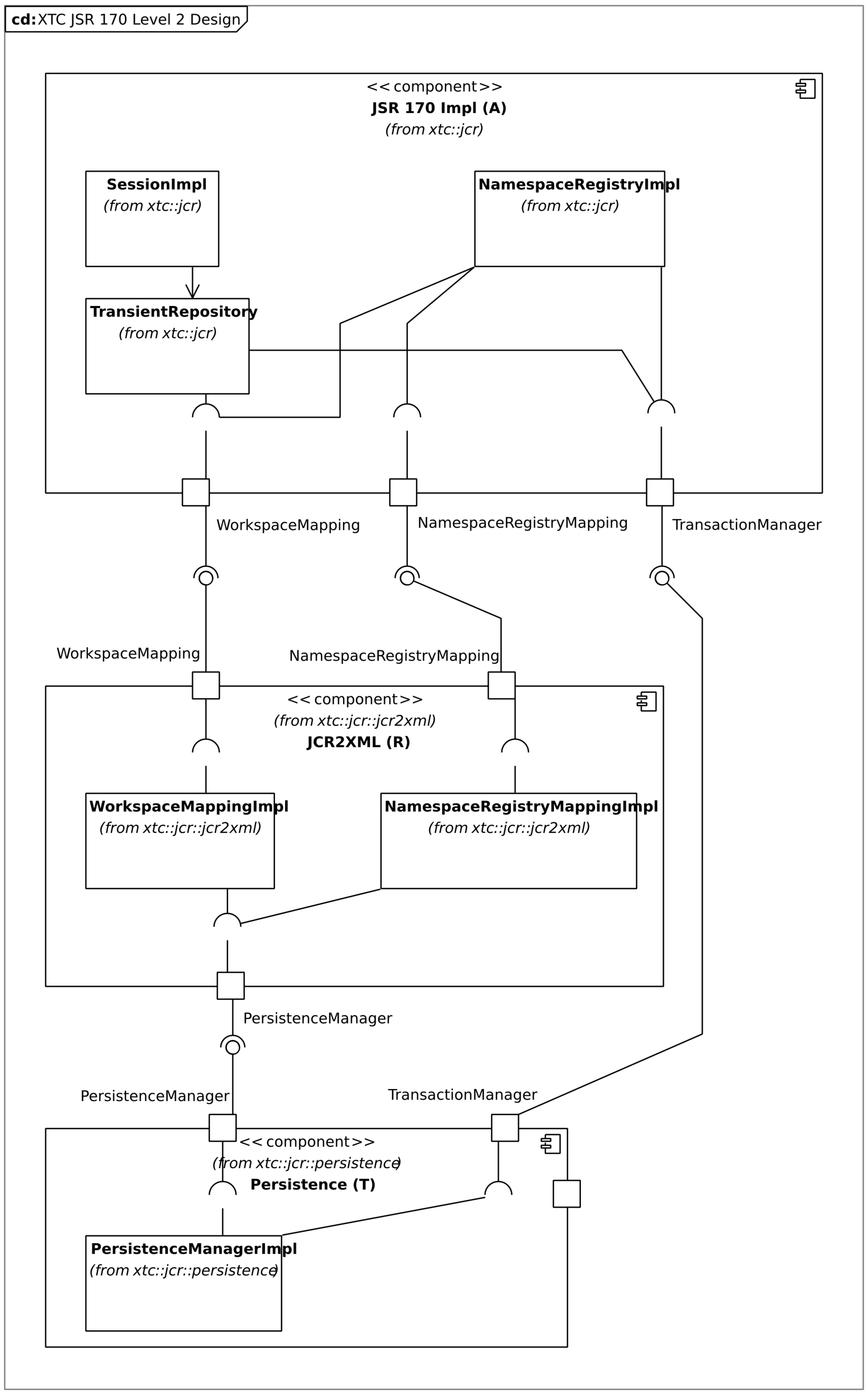


Figure 4.8: XTC JCR *Level 2* Design

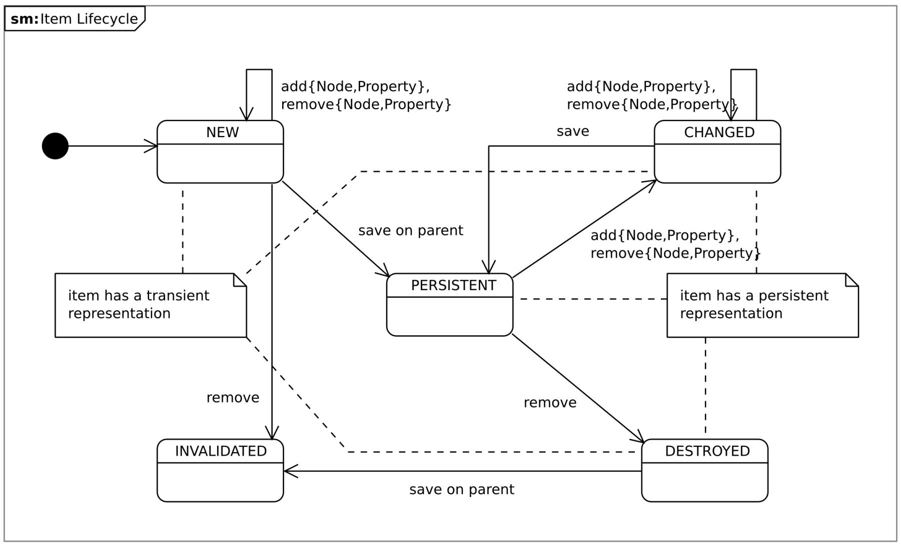


Figure 4.9: Item Lifecycle

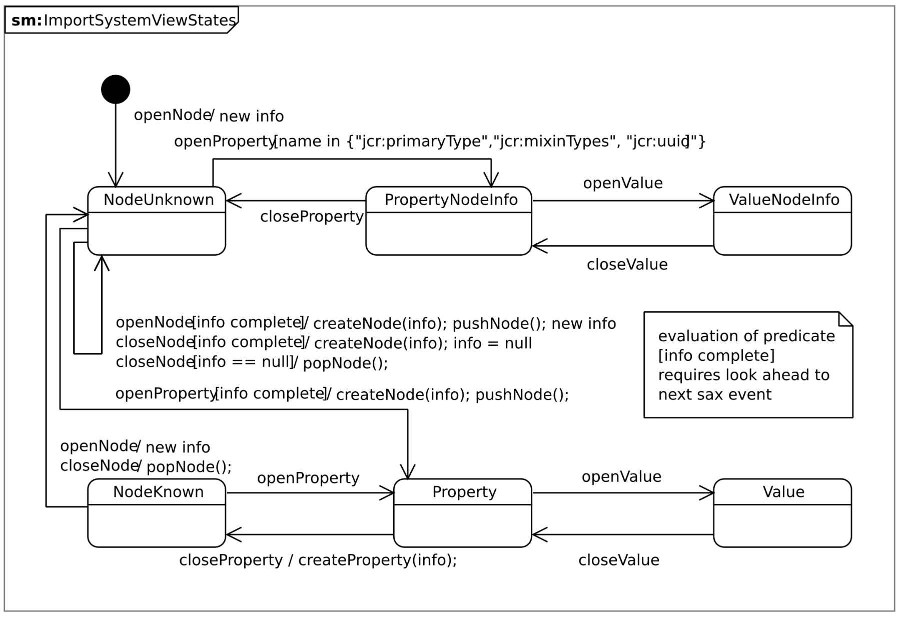

sm:ImportSystemViewStates
openNode/ new info
openProperty[name in {"jcr:primaryType","jcr:mixinTypes", "jcr:uuid"}
NodeUnknown
closeProperty
PropertyNodeInfo
openValue
ValueNodeInfo
closeValue
openNode[info complete]/ createNode(info); pushNode(); new info
closeNode[info complete]/ createNode(info); info = null
closeNode[info == null]/ popNode();
openProperty[info complete]/ createNode(info); pushNode();
evaluation of predicate [info complete] requires look ahead to next sax event
openNode/ new info
closeNode/ popNode();
NodeKnown
openProperty
Property
openValue
Value
closeProperty / createProperty(info);
closeValue

Figure 4.10: States of the *systemview* importer

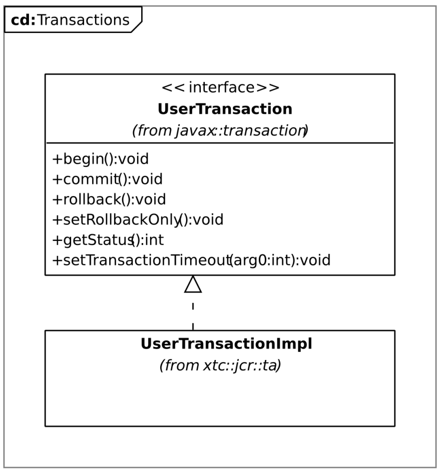


Figure 4.11: JTA UserTransaction implementation in XTC JCR

4.3 Transactions

The way the JCR interface is designed synchronization problems can arise when the repository is modified concurrently. *Lost Update* and *Inconsistent Read* are possible. In order to avoid these problems the specification proposes transaction support as an optional feature. Instead of specifying another transaction API the JSR 170 delegates to the *Java Transaction API* (JTA) [CM02] specification. JTA stipulates a *UserTransaction* interface that allows for user managed transactions, i.e. the client application controls the transactional behavior. It, furthermore, specifies how distributed transactions are handled in a container managed environment, such as a Java enterprise application server.

XTC does not yet provide a JTA interface. The current driver interface supports *begin*, *commit*, and *rollback* of a transaction. Yet it lacks a *prepare* method which would be vital to implement a two phase commit protocol as specified for distributed transactions. Therefore, only the simpler *UserTransaction* interface (see figure 4.11) is implemented. This implementation itself is straight forward. The required methods are delegated directly to the driver interface.

4.4 Versioning

Versioning is an optional feature in the JSR 170 specification. The *Versioning* component in a JCR allows state changes of workspace nodes to be recorded as versions in a common repository version storage. At a later point, JCR users can browse and restore these versions. The versioning system is modeled after the *Workspace Versioning and Configuration Management* (WVCM) API defined by JSR 147 [Nue06, Cle07].

The component in a repository where all version data is stored is called *version storage*. Each versionable node, determined by the presence of mixin type *mix:versionable*, has a *versionHistory* in the *version storage* under which all

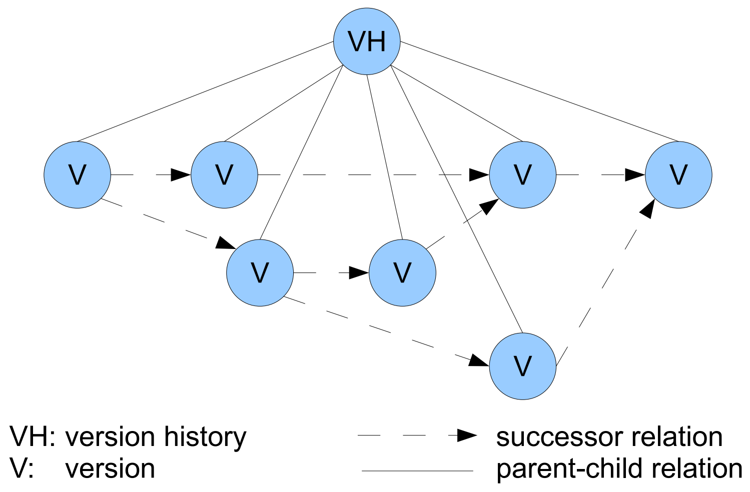


Figure 4.12: Version history of one node and its successor relation

versions of this node are stored. There is only one central *version storage* instance for all workspaces. The reason why the *version storage* is common to all workspaces becomes apparent in repositories that support multiple workspaces, like XTC JCR. Nodes can have multiple representations in different workspaces, affiliated by a common UUID. These representations represent branched versions of the same node, and thus they share a common version history across workspaces.

Among these versions predecessor and successor relations are kept. These relations form acyclic directed graphs (see figure 4.12).

The versioning system introduces a *checkout* / *checkin* mechanism. Versionable nodes and their non-versionable subtree[11] have read-only protection until they are *checked out*. Once all modifications on a node are made persistent via a *save* call, the client calls *checkin* and a new version is created. During *checkin* the read-only state is automatically restored. Note that the read-only mechanism inhibits concurrent modifications on a node within a workspace.

What is stored in a version depends on the node type of the versionable node and its non-versionable subtree. More precisely, every child node and property definition specifies the version behavior via the *OnParentVersion* attribute. This attribute determines the behavior when the parent node is checked in. The attribute value is one of the following constants (see [Nue06] chap. 8.2.11) :

COPY This and all descendent items are copied to the version storage.

VERSION For properties and non-versionable child nodes VERSION has the same effect as COPY. For a versionable child node a reference to its version history is stored and recursion in the subtree stops.

[11] A versionable node can have versionable and non-versionable descendants. All versionable descendants have their own *versionHistory* and are handled individually by the versioning system. Thus, for each versionable node the versioning system must manage the non-versionable part of its subtree.

INITIALIZE A new item will be created in the version. This is ignored in the reference implementation and XTC JCR. (same as IGNORE)

COMPUTE A new item will be computed in the version. This is ignored in the reference implementation and XTC JCR. (same as IGNORE)

IGNORE The item is simply skipped and not versioned.

ABORT On presents of such a child item checkin is aborted.

The use of VERSION results in incremental versioning, while COPY creates full and redundant copies.

The version storage is modeled as special JCR workspace, called *versionStorage*. This enables reuse of the type system, the query engine, and workspace mappings.

Each workspace session has access to an exclusive *versionStorage* session which is instantiated in parallel. However, the second session runs in the same persistence context, sharing the same XTC connection (*PersitenceManager*) and XTC transaction (*TransactionManger*).

The version storage must be accessible from each workspace under */jcr:system/jcr:versionStorage*. The direct access methods and query processing delegate transparently to the central *versionStorage* workspace when needed.

The system is equipped with a *Versioning* component. This component handles the "direct to workspace" *checkout* / *checkin* mechanism and implements the read-only protection (*isCheckedOut*) for versioned nodes and their non-versionable subtree.

By definition of the mixin type *mix:versionable* each versionable node has a property *isCheckedOut*. The versioning implementation reads and sets this property directly on the workspace, without going through the transient layer. When this flag is set to *false* no modification is performed on this node. Protected by an XTC transaction this flag realizes the global read-only protection. In this case, the specification also requires all non-versionable descendants to be read-only. For each modification on a non-versionable node the read-only check recurses up the path towards the root node. The path must be checked for the next versionable parent and its *isCheckedOut* property. This sounds expensive, but in practice the following optimization rule applies.

If this check encounters a modified parent P the check can terminate, no matter if P is versionable or not. This is why: If there is no versionable node on the root path, this optimization is correct. No read-only protection applies at all. If there exists a versionable node on the root path, let V be the first versionable parent of P. P is equal to V or P is in the subtree of V. In both cases V must have been checked out prior to P's modification and cannot yet be checked in, since prior to a *checkin* all modifications in the subtree must be persisted.

Up to now the implementation supports read-only protection, creation of versions, and browsing of the version graph. *Restore* and several cross workspace methods, such as *merge*, are not implemented yet, due to the lack of time.

Chapter 5

Evaluation

5.1 Demo Applications

Two demo applications were developed. Prior to implementation of XTC JCR they were built using the reference implementation as a JCR back-end. XTC JCR was then used as a drop in replacement for Jackrabbit.

5.1.1 Deployment on Glassfish

Web applications are one important field of application for *Java Content Repositories*. It is therefore natural that both demo applications are web applications as well. In the test setup these applications are run on the open-source application server *Glassfish*[1].

The reference implementation, Jackrabbit, and XTC JCR are registered as JNDI resources. This way the JCR back-end of the demo applications can be switched transparently.

XTC JCR only has very few external dependencies. In order to run it on the server the following libraries must be put into the application server's classpath.[2]

PWD_Common.jar Common utilities for String manipulation.

xercesImpl.jar Contains required XMLChar class.

XTCdriver.jar The XTC driver.

jcr-1.0.jar The JCR API.

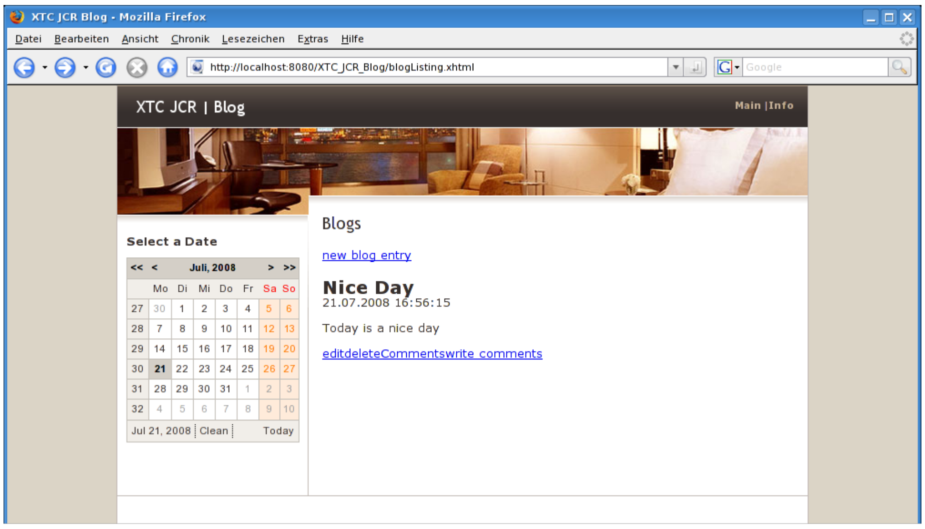
XTC_JCR.jar The XTC JCR implementation.

For instructions on how to deploy Jackrabbit please see the online manual at the project's official website[3].

[1] `https://glassfish.dev.java.net/`

[2] On Linux it is sufficient to symlink these jar files into the domains library folder.

[3] `http://jackrabbit.apache.org/`


XTC JCR Blog - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://localhost:8080/XTC_JCR_Blog/blogListing.xhtml
Google
XTC JCR | Blog
Main |Info
Select a Date
<< < Juli, 2008 > >>
Mo Di Mi Do Fr Sa So
Jul 21, 2008 | Clean | Today
Blogs
new blog entry

Nice Day
21.07.2008 16:56:15

Today is a nice day

editdeleteCommentswrite comments

Figure 5.1: Screenshot of the Blog Demo Application (using JBoss Seam)

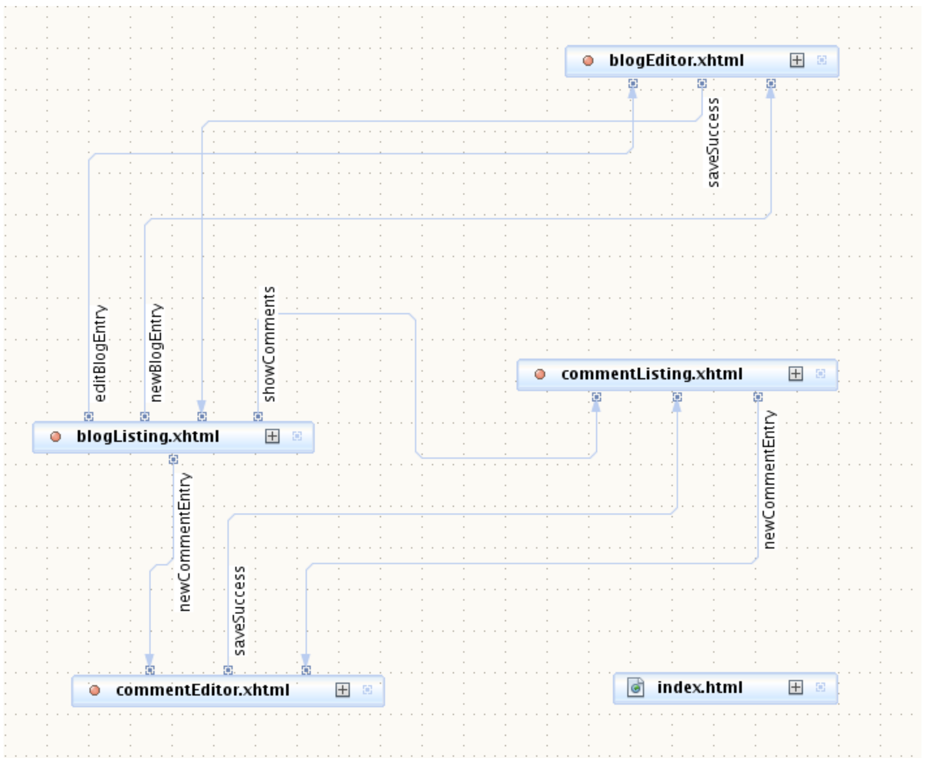
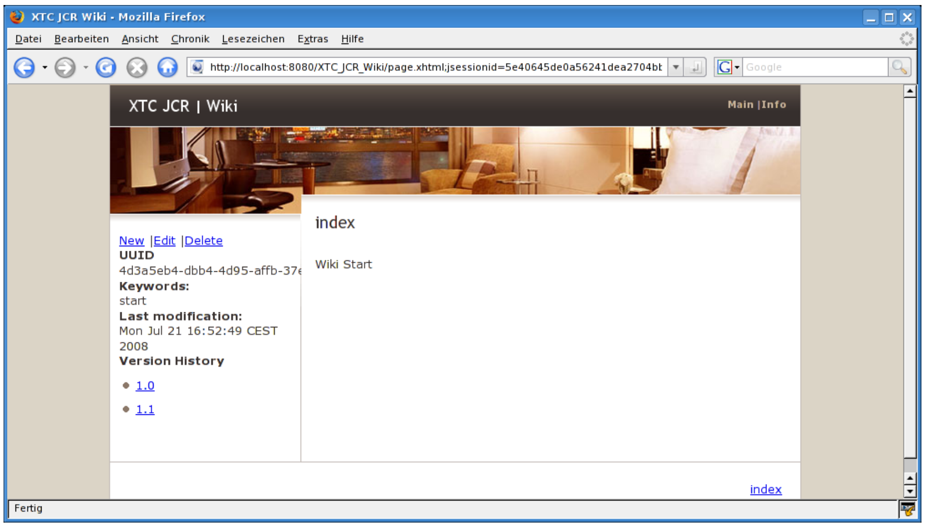


Figure 5.2: JSF page flow of Blog Demo Application


XTC JCR Wiki - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://localhost:8080/XTC_JCR_Wiki/page.xhtml;jsessionid=5e40645de0a56241dea2704bt
Google
XTC JCR | Wiki
Main |Info
New |Edit |Delete
Fertig

UUID
4d3a5eb4-dbb4-4d95-affb-37e
Keywords:
start
Last modification:
Mon Jul 21 16:52:49 CEST 2008
Version History

- 1.0
- 1.1

index

Wiki Start

index

Figure 5.3: Screenshot of the Wiki Demo Application (using JBoss Seam)

The first demo application is a simple blogging system (see figure 5.1). It allows the blogger to publish an article. Other users are able to comment on the article (see figure 5.2). This application demonstrates the basic *Level 1* and *Level 2* capabilities of a *Java Content Repository*: browsing and editing content. The second demo application is a simple form of a wiki system (see figure 5.3). It allows articles to be created and edited, as well. Its main purpose is to demonstrate a JCR's optional feature "Versioning". For each page a version history is kept—a central feature in a wiki system. Via the version history (see screenshot 5.3) users can access all past versions of an article, a protection against intentional or non-intentional fraud.

5.2 Performance

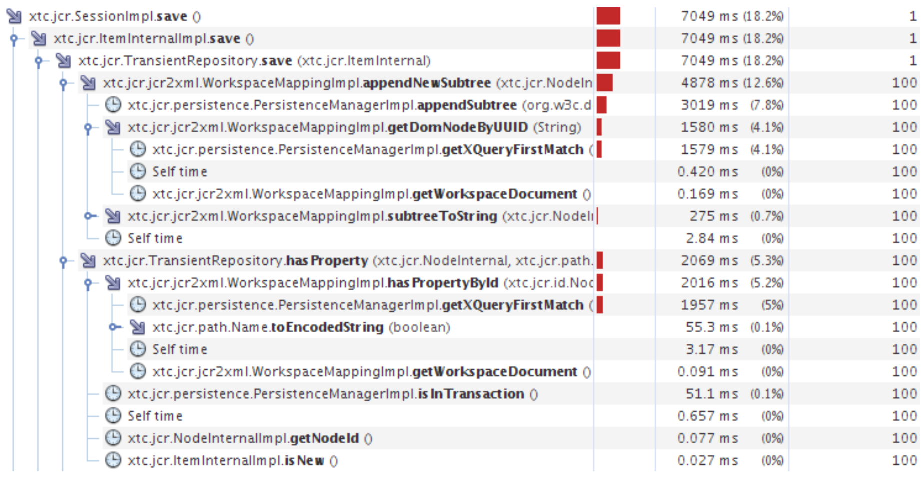
In order to evaluate performance, XTC JCR was benchmarked against Jackrabbit RI version 1.4. XTC JCR is designed to run on distributed clients accessing XTC as the central database. The reference implementation supports different deployment models. In order to obtain fair results, the reference JCR must also support multiple distributed clients. Deployment model number three, "Repository Server"[4], is the only feasible model in this context. A *Model 3* deployment runs Jackrabbit's repository instance as a standalone server. All clients connect to the server via network (RMI, WebDav, etc.).

The test environment includes XTC and XTC JCR on the one hand, a Jackrabbit RMI server backed by a PostgreSQL[5] database on the other.

The tests were designed to investigate the performance of content modifica-

[4] The other Jackrabbit deployment models are: "Application Bundle" (*Model 1*), "Shared J2EE Resource" (*Model 2*)

[5] `http://www.postgresql.org`



```
xtc.jcr.SessionImpl.save ()                                                          7049 ms (18.2%)    1
  xtc.jcr.ItemInternalImpl.save ()                                                   7049 ms (18.2%)    1
    xtc.jcr.TransientRepository.save (xtc.jcr.ItemInternal)                          7049 ms (18.2%)    1
      xtc.jcr.jcr2xml.WorkspaceMappingImpl.appendNewSubtree (xtc.jcr.NodeIn          4878 ms (12.6%)  100
        xtc.jcr.persistence.PersistenceManagerImpl.appendSubtree (org.w3c.d          3019 ms  (7.8%)  100
        xtc.jcr.jcr2xml.WorkspaceMappingImpl.getDomNodeByUUID (String)               1580 ms  (4.1%)  100
          xtc.jcr.persistence.PersistenceManagerImpl.getXQueryFirstMatch (           1579 ms  (4.1%)  100
          Self time                                                                 0.420 ms    (0%)  100
          xtc.jcr.jcr2xml.WorkspaceMappingImpl.getWorkspaceDocument ()              0.169 ms    (0%)  100
        xtc.jcr.jcr2xml.WorkspaceMappingImpl.subtreeToString (xtc.jcr.Nodel           275 ms  (0.7%)  100
        Self time                                                                   2.84 ms    (0%)  100
      xtc.jcr.TransientRepository.hasProperty (xtc.jcr.NodeInternal, xtc.jcr.path.   2069 ms  (5.3%)  100
        xtc.jcr.jcr2xml.WorkspaceMappingImpl.hasPropertyById (xtc.jcr.id.Noc         2016 ms  (5.2%)  100
          xtc.jcr.persistence.PersistenceManagerImpl.getXQueryFirstMatch (           1957 ms    (5%)  100
          xtc.jcr.path.Name.toEncodedString (boolean)                                55.3 ms  (0.1%)  100
          Self time                                                                  3.17 ms    (0%)  100
          xtc.jcr.jcr2xml.WorkspaceMappingImpl.getWorkspaceDocument ()              0.091 ms    (0%)  100
        xtc.jcr.persistence.PersistenceManagerImpl.isInTransaction ()                51.1 ms  (0.1%)  100
        Self time                                                                   0.657 ms    (0%)  100
        xtc.jcr.NodeInternalImpl.getNodeId ()                                       0.077 ms    (0%)  100
        xtc.jcr.ItemInternalImpl.isNew ()                                           0.027 ms    (0%)  100
```

Figure 5.4: *addNode100* in Netbeans Profiler

tion, direct data access, and declarative data retrieval. Each test was run five times. The results presented here are the averaged test duration measurements. A Lenovo T60[6] Laptop computer running Java[7] version 1.6 served as testing platform.

The first benchmarking results were really disappointing [8]. Netbeans integrated profiler helped to find the bottlenecks in the XTC JCR code and to optimize the system.

Optimization	Description
No Opt.	Initial version without optimization.
Cache NS Map	Caching of namespaces (prefix to URI mapping) in each transaction.
Cache WS & NS Doc	Caching of workspace and namespace DOM document in each transaction.
Long TA	Long transaction for the time of *addNode* operation.
Div. Opt.	Several small optimizations in methods: *getPrefix*, *getURI*, and *hasProperty*.
XTC Caching	Meta data caching in XTC enabled.
NodeTypeManager Opt.	Item types are constant in JSR 170. Computed result (e.g. matching node types) can be cached.

Table 5.1: Optimizations for data modifications.

One major problem is *Remote Method Invocation* (RMI) overhead. XTC JCR runs in the client JVM. Performance suffers a great deal when many RMI calls to the XTC system are required in order to fulfill the client's request. Each RMI call comes with an additional overhead for transmitting data over the network and is significantly slower than a local method call. The way JSR 170

[6] Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz, 2GB RAM; x86_64 Linux 2.6.25-gentoo-r5

[7] Java(TM) SE Runtime Environment (build 1.6.0_ 07-b06) Java HotSpot(TM) 64-Bit Server VM (build 10.0-b23, mixed mode)

[8] 26 seconds (!) for the first run of test *addNode100*, see 5.5

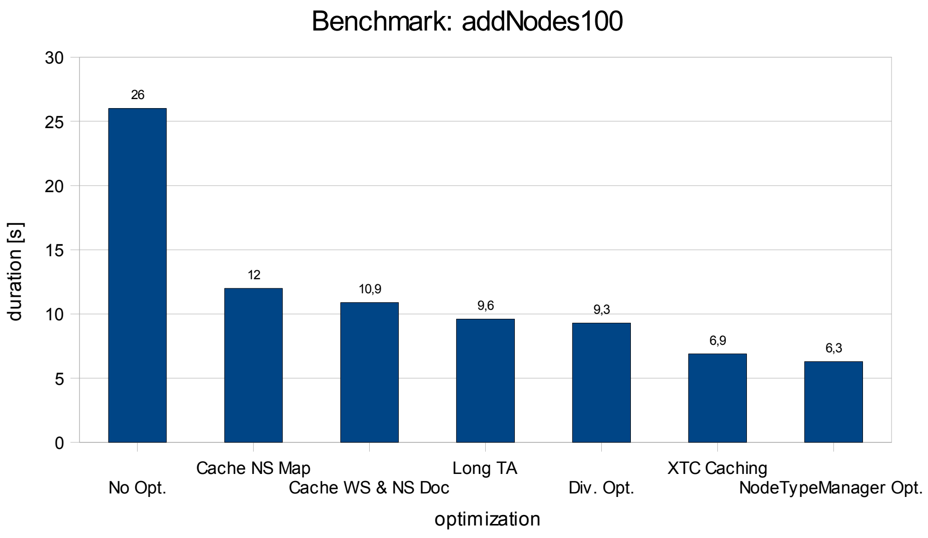


Figure 5.5: Optimization steps for benchmark *addNode100* (persisting 100 new nodes under a persistent node)

is designed, some of these remote calls cannot be avoided. JCR namespaces are potentially instable, due to the ability to remap namespaces. Only for the time of a transaction can the system assume that the namespace mapping is constant. However, JCR operations are not required to be enclosed by a global transaction. Remapping of a namespace prefix is a very uncommon operation, but still, in the worst case XTC JCR must query the database for the current namespace mapping in each JCR operation.

A series of optimization steps (see figure 5.5 and explanation in table 5.1) helped to alleviate this effect. Network latency is a typical problem in distributed applications. The typical solution is caching. Caching of namespace mappings and DOM documents (version storage document, workspace document, and namespace document) for the time of one transaction allowed to reduce the time for data modification tests by over 75%.

Further optimizations in the code target the minimization of network traffic. When designing interfaces for remote operation, it is good practice to combine all required data as parameters in a specially designed method signature, over setting parameters via several invocations of more general remote methods. The same holds for the return type of the remote operation. It is more efficient to transfer the required data at once than calling simple getter-methods on a remote result object. However, this requires the request to be specifically tailored to the required information.

The early XTC JCR implementation used XQuery statements to retrieve DOM nodes. Retrieving a child node or an attribute value of a result node requires RMI calls again. During profiling it turned out that it is more efficient to tailor the XQuery statement specifically to the required data. Fetching the result as serialized string is faster than using DOM operations on the result nodes.

As a third improvement DOM queries were substituted with calls to a brand new API of the XTC system. *Java Specification Request* 225 [Mel07] defines the *XQuery API for Java* (XQJ). XTC supports an XQJ-like interface which allows to avoid parsing of the result string.

The *getPrefix* method of the namespace registry serves as a good example. The first version (see simplified code in listing 5.1) finds the DOM node representing the prefix to URI mapping in the namespace registry document. In a second RMI call the prefix attribute value is returned. The improved version (see simplified code in listing 5.2) uses a FLWOR XQuery statement to return the required data directly.

```java
String query = "doc(\"namespaceRegistry.xml\")/registry/namespace[@uri=\"" + uri + "\"][1]";
// first RMI call
Node node = this.persistenceManager.getXQueryFirstMatch(this.getNamespaceRegistryDocument(), query);

if(node == null) return null;
// second RMI call
return this.persistenceManager.getAttributeValue(node, PREFIX);
```

Listing 5.1: *getPrefix(String uri)* first version (DOM)

```java
String query = "for␣$v␣in␣doc(\"namespaceRegistry.xml\")/registry/namespace[@uri=\""+uri+"\"][1]/@prefix␣return␣<n␣prefix=\"{fn:data($v)}\"/>";

Sequence sequence = this.persistenceManager.getXQueryJSR225(query);

for(Item i : sequence){
  if(i.getType() == ItemType.NODETYPE_ATTRIBUTE){
    return i.getString();
  }
}
return null;
```

Listing 5.2: *getPrefix(String uri)* improved version (XQJ)

The *getPrefix* and *getURI* methods were accelerated by 70% from 17.85ms to 5.23ms on average. In this fashion the implementations of the workspace and namespace mapping layers were revised. Queries were rewritten to take advantage of XQJ. XTC's query processor was extended by functions *fn:node-name* and *jcr:depth* to allow faster computation in the server with less network overhead. As a result the *getNodeByNodeId* query was accelerated by 82% from 101ms to 18ms on average.

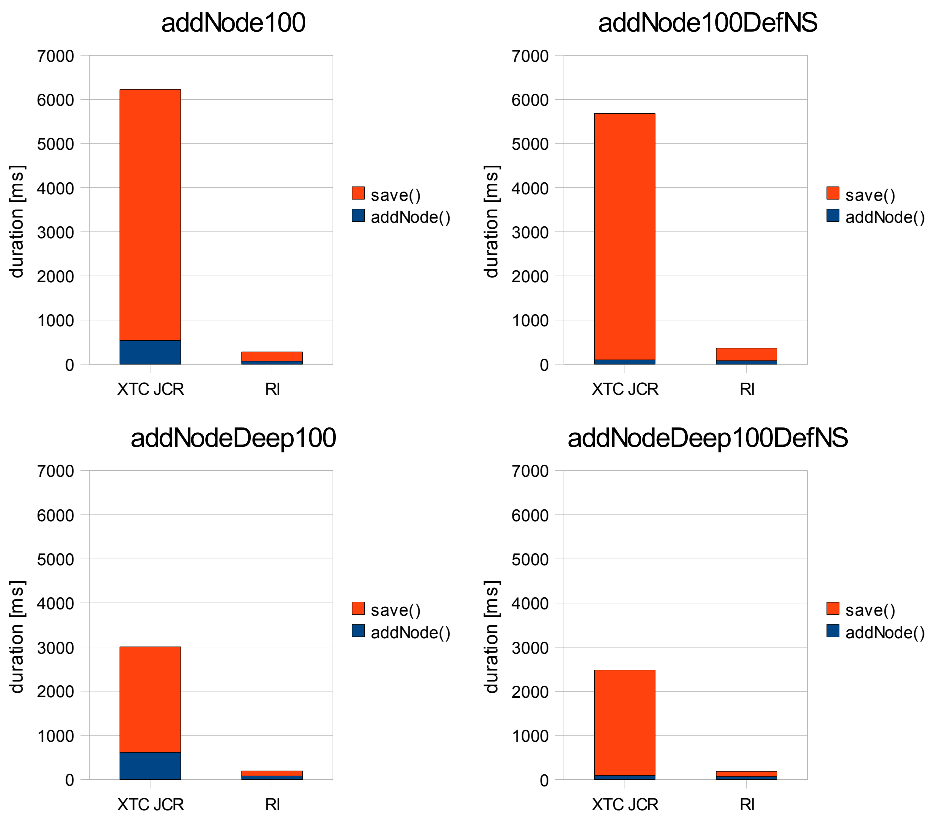


Figure 5.6: Data Modification Benchmarks

Test	Description
addNode100	Persists 100 nodes as direct child nodes of a persistent node. This results in 100 append operations in XTC.
addNode100DefNS	Same as *addNode100*, but the names of the new nodes do not carry a namespace prefix.
addNodeDeep100	Persists 100 nodes under a persistent node. Each new node is as a child of the previously added node. All 100 nodes are appended as a serialized string. This results in a single subtree add operation in XTC.
addNodeDeep100DefNS	Same as *addNodeDeep100*, but the names of the new nodes do not carry a namespace prefix.

Table 5.2: Test suite: data modification

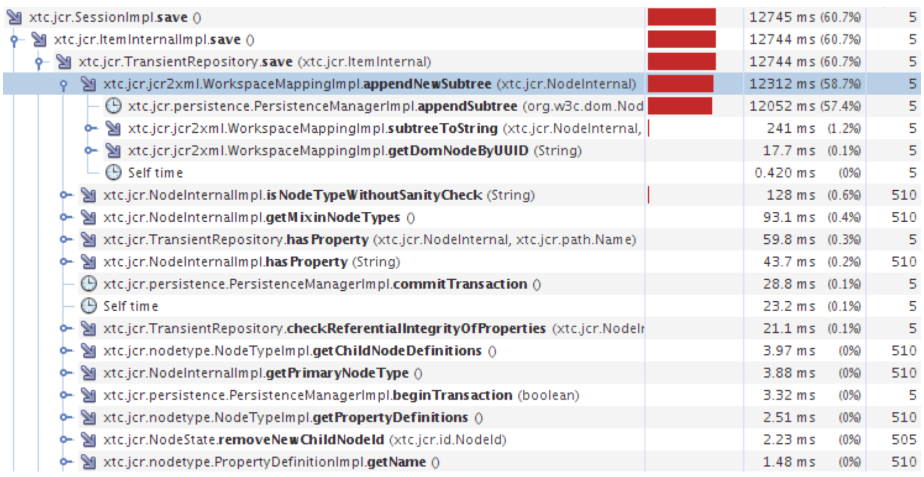
5.2.1 Data Modification

AddNode100 is one of four tests, designed to measure data modification performance (see table 5.2). The results in figure 5.6 show how instable namespace mappings influence the overall performance of XTC JCR. Everytime a prefix must be resolved an expensive RMI call to XTC must be made. The comparison of the test run with qualified names (on the left) to the test run with unprefixed names (on the right) demonstrates the cost of prefix to URI resolvation in each transaction. The test shows also that the *addNode* operation called with an unprefixed name is of equal speed to the RI pendant (figure 5.6 *addNode100DefNS*).

In XTC JCR the major part of the time is spent in the *save()* method. This is only partly due to the RMI overhead, as described previously. The impact of RMI overhead can be quantified when comparing *addNode100(DefNS)* to *addNodeDeep100(DefNS)* (top to bottom). Test *addNode100* uses 100 *appendSubtree* RMI calls to append one JCR node each call. Test *addNodeDeep100* uses only one *appendSubtree* invocation to append the whole subtree of 100 serialized JCR nodes. The difference of approximately three seconds makes up about 50% of the total time.

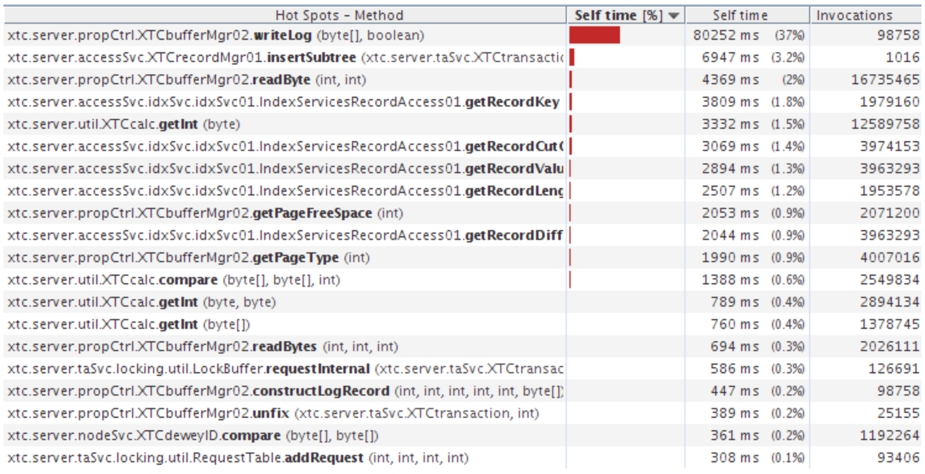
Looking at profiling report of *addNode100Deep* (figure 5.7) it becomes clear that most of the time is spent during the *appendSubtree* execution which delegates directly to the remote operation in XTC. During a longer running profiling session of XTC (see profiling report figure 5.8) the *writeLog()* method was identified as a bottleneck. It turns out that writing the log entries for the *insertSubtree* method is more than ten times slower than the actual *insertSubtree* operation. XTC currently uses *physical* logging to ensure ACID properties. This logging strategy copies a page to the log on each modification within the page. This is slow since it results in a lot of IO overhead. Currently, the implementation of a new *physiological* logging system is in progress. *Physiological* logging saves snapshots of pages and only logs logical operations on the data. It will result in a faster *writeLog()* operation.

The profiling report 5.4 of the *addNode100* test reveals more potential for performance improvements. Test *addNode100* invokes the query in *getDomNodeByUUID* 100 time which takes up 1.5 seconds. Getting the DOM node by its



xtc.jcr.SessionImpl.**save** ()		12745 ms (60.7%)	5
xtc.jcr.ItemInternalImpl.**save** ()		12744 ms (60.7%)	5
xtc.jcr.TransientRepository.**save** (xtc.jcr.ItemInternal)		12744 ms (60.7%)	5
xtc.jcr.jcr2xml.WorkspaceMappingImpl.**appendNewSubtree** (xtc.jcr.NodeInternal)		12312 ms (58.7%)	5
xtc.jcr.persistence.PersistenceManagerImpl.**appendSubtree** (org.w3c.dom.Nod		12052 ms (57.4%)	5
xtc.jcr.jcr2xml.WorkspaceMappingImpl.**subtreeToString** (xtc.jcr.NodeInternal,		241 ms (1.2%)	5
xtc.jcr.jcr2xml.WorkspaceMappingImpl.**getDomNodeByUUID** (String)		17.7 ms (0.1%)	5
Self time		0.420 ms (0%)	5
xtc.jcr.NodeInternalImpl.**isNodeTypeWithoutSanityCheck** (String)		128 ms (0.6%)	510
xtc.jcr.NodeInternalImpl.**getMixinNodeTypes** ()		93.1 ms (0.4%)	510
xtc.jcr.TransientRepository.**hasProperty** (xtc.jcr.NodeInternal, xtc.jcr.path.Name)		59.8 ms (0.3%)	5
xtc.jcr.NodeInternalImpl.**hasProperty** (String)		43.7 ms (0.2%)	510
xtc.jcr.persistence.PersistenceManagerImpl.**commitTransaction** ()		28.8 ms (0.1%)	5
Self time		23.2 ms (0.1%)	5
xtc.jcr.TransientRepository.**checkReferentialIntegrityOfProperties** (xtc.jcr.NodeIr		21.1 ms (0.1%)	5
xtc.jcr.nodetype.NodeTypeImpl.**getChildNodeDefinitions** ()		3.97 ms (0%)	510
xtc.jcr.NodeInternalImpl.**getPrimaryNodeType** ()		3.88 ms (0%)	510
xtc.jcr.persistence.PersistenceManagerImpl.**beginTransaction** (boolean)		3.32 ms (0%)	5
xtc.jcr.nodetype.NodeTypeImpl.**getPropertyDefinitions** ()		2.51 ms (0%)	510
xtc.jcr.NodeState.**removeNewChildNodeId** (xtc.jcr.id.NodeId)		2.23 ms (0%)	505
xtc.jcr.nodetype.PropertyDefinitionImpl.**getName** ()		1.48 ms (0%)	510

Figure 5.7: *addNode100Deep* in Netbeans Profiler



Hot Spots - Method	Self time [%]	Self time	Invocations
xtc.server.propCtrl.XTCbufferMgr02.**writeLog** (byte[], boolean)		80252 ms (37%)	98758
xtc.server.accessSvc.XTCrecordMgr01.**insertSubtree** (xtc.server.taSvc.XTCtransacti		6947 ms (3.2%)	1016
xtc.server.propCtrl.XTCbufferMgr02.**readByte** (int, int)		4369 ms (2%)	16735465
xtc.server.accessSvc.idxSvc.idxSvc01.IndexServicesRecordAccess01.**getRecordKey**		3809 ms (1.8%)	1979160
xtc.server.util.XTCcalc.**getInt** (byte)		3332 ms (1.5%)	12589758
xtc.server.accessSvc.idxSvc.idxSvc01.IndexServicesRecordAccess01.**getRecordCut(**		3069 ms (1.4%)	3974153
xtc.server.accessSvc.idxSvc.idxSvc01.IndexServicesRecordAccess01.**getRecordValu**		2894 ms (1.3%)	3963293
xtc.server.accessSvc.idxSvc.idxSvc01.IndexServicesRecordAccess01.**getRecordLeng**		2507 ms (1.2%)	1953578
xtc.server.propCtrl.XTCbufferMgr02.**getPageFreeSpace** (int)		2053 ms (0.9%)	2071200
xtc.server.accessSvc.idxSvc.idxSvc01.IndexServicesRecordAccess01.**getRecordDiff**		2044 ms (0.9%)	3963293
xtc.server.propCtrl.XTCbufferMgr02.**getPageType** (int)		1990 ms (0.9%)	4007016
xtc.server.util.XTCcalc.**compare** (byte[], byte[], int)		1388 ms (0.6%)	2549834
xtc.server.util.XTCcalc.**getInt** (byte, byte)		789 ms (0.4%)	2894134
xtc.server.util.XTCcalc.**getInt** (byte[])		760 ms (0.4%)	1378745
xtc.server.propCtrl.XTCbufferMgr02.**readBytes** (int, int, int)		694 ms (0.3%)	2026111
xtc.server.taSvc.locking.util.LockBuffer.**requestInternal** (xtc.server.taSvc.XTCtransac		586 ms (0.3%)	126691
xtc.server.propCtrl.XTCbufferMgr02.**constructLogRecord** (int, int, int, int, int, byte[]		447 ms (0.2%)	98758
xtc.server.propCtrl.XTCbufferMgr02.**unfix** (xtc.server.taSvc.XTCtransaction, int)		389 ms (0.2%)	25155
xtc.server.nodeSvc.XTCdeweyID.**compare** (byte[], byte[])		361 ms (0.2%)	1192264
xtc.server.taSvc.locking.util.RequestTable.**addRequest** (int, int, int, int)		308 ms (0.1%)	93406

Figure 5.8: Hotspot analysis of XTC's *appendSubtree* operation in Netbeans Profiler

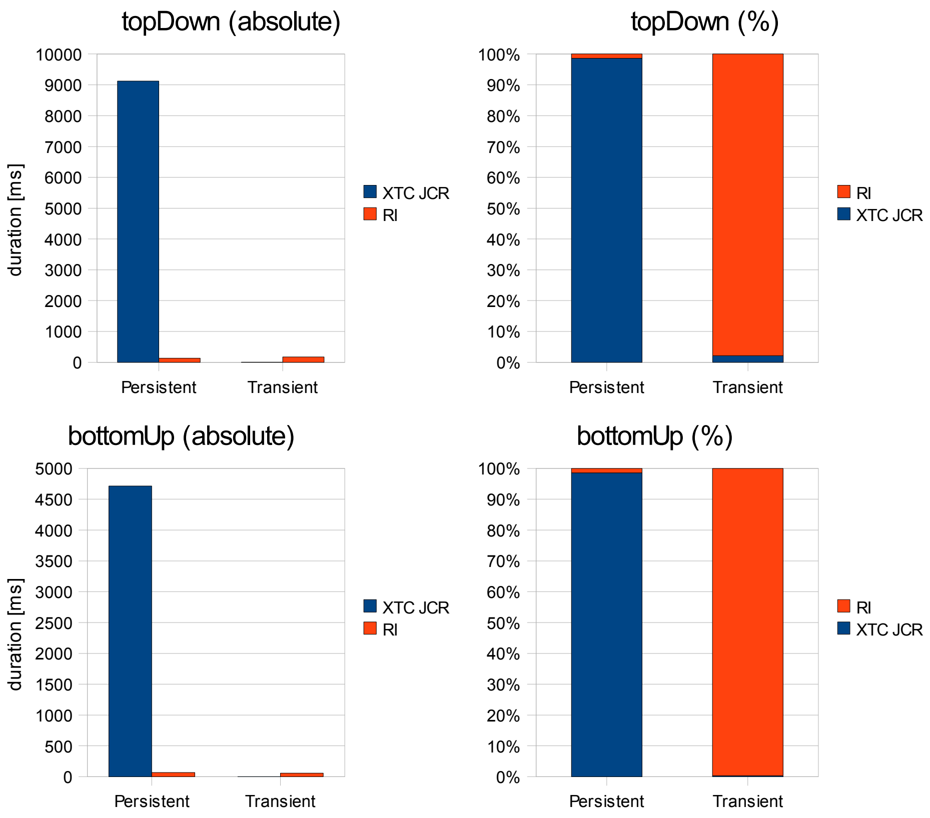


Figure 5.9: Navigational Access Benchmarks

UUID is a very common operation, since all data editing must still be made via the DOM interface. The query in *hasPropertyById* takes up 2 seconds. Prior to insertion of a new node the system must check for a conflicting property node with the same name as the node. Currently, index support is not yet available. With index support enabled the time for these operations is expected to drop dramatically. An index for all *xtcjcr:uuid* attributes will allow the query processor to find a node by its UUID much quicker. Index support is actually a major feature that the whole XTC JCR design relies on. It is expected to speed up almost all queries and thus the implementation as a whole. At the time of writing index integration into the XQuery processor and into the data manipulation system is in progress.

5.2.2 Navigational Access

Navigational access was tested using two tests described in table 5.3. These tests are designed to measure direct access speed by traversing through the JCR content tree of depth 101 top to bottom and vice versa. The results in figure 5.9 show that XTC JCR is slower on persistent nodes, while it is faster, by about the same magnitude, on transient nodes compared to the RI. Once index support is available the persistent results are expected to improve drastically,

Test	Description
topDown	100 *Node.getNode* (returns a child node) invocations (run on a transient and a persistent content tree)
bottomUp	100 *Node.getParent* (returns the parent node) invocations (run on a transient and a persistent content tree)

Table 5.3: Test suite: navigational access

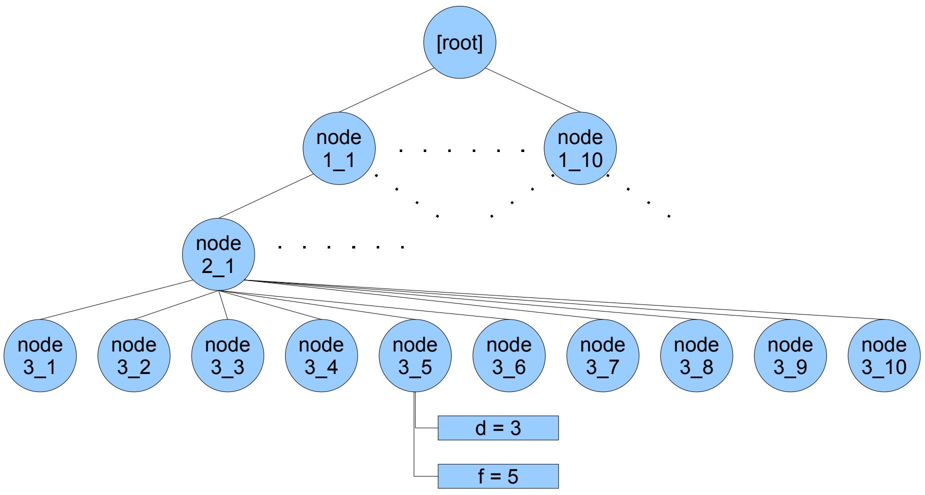


Figure 5.10: Declarative Query Test Data

since finding a node by its UUID is crucial to both *getNode* and *getParent*.

Furthermore, one can observe that the reference implementation shows similar access speeds for persistent and transient access. This is due to caching mechanisms. In Jackrabbit data modifications are directly written to the persistent back-end. However, the content itself stays cached and can quickly be retrieved without employing the database at all. Jackrabbit assumes hereby that it has exclusive write access to the database.

5.2.3 Declarative Query

In order to test query performance of a JCR implementation a suite of ten queries (see table 5.4) is run on a workspace filled with test data. The test data consists of a ten-ary[9] tree of depth three (see figure 5.10). Each node has two properties: depth $d$ and fan $f$. Depth $d$ holds an integer value representing the depth level of the current node. Fan $f$ is an integer property valued from one to ten, designed as a simple index for each child node. The node names are constructed as *node$d$_$f$*.

XTC JCR is a little slower when data is accessed direcly via the content structure as in queries number one and two. Queries number three through eight show

[9] n-ary trees are trees where each node has n child nodes.

no.	Query	result size	XTC JCR [ms]	RI [ms]
1	*//node3_1*	100	72.0	30.8
2	*//jcr:root/queryTest//node3_2*	100	130.6	32.8
3	*//*[@d = 3 and @f = 2]*	100	3104.4	70.2
4	*//*[@d = 1]//*[@d = 3]*	1000	4158.6	491.4
5	*//*[*/@d = 3]*	100	3443.6	29.2
6	*//*[@d = 1]/*[@d = 2]/*[@d = 3]*	1000	3349.2	262.4
7	*//*[@d >= 1]/*[@d >= 2]/*[@d >= 3]*	1000	3312.2	234.0
8	*//*[@d >= 2]/*[@d >= 3]*	1100	3305.6	228.8
9	*//*[node3_1]*	100	1287.4	n.a.
10	*//*[node3_1 and node3_2]*	100	1466.0	n.a.

Table 5.4: Test suite: query performance

how the reference implementation profits from its Lucene[10] index. As soon as XTC's index support and holistic twig join [Hüh08] support is available to the XQuery processor these results are expected to change in favor of XTC JCR.

Queries nine and ten cannot be handled by the current reference implementation. Query nine returns after about 500ms and query ten computes for several minutes. Both queries return the erroneous result size of -1. XTC JCR returns correct results in a reasonable amount of time.

5.2.4 Concurrent Modifications

All prior tests have been run without using optional feature "Transactions". Run with a single global transaction for each test, XTC JCR performs *addNode100*, *addNode100Deep*, *topDown* (Persistent), and *bottomUp* (Persistent) about 500ms faster. In these cases the implementation profits of caching and the reduced overhead for opening and closing many small transactions. The other tests are not affected significantly.

XTC is designed to handle concurrent read and write access on XML data well. In a last test the performance of concurrent access by multiple clients to the repository was to be tested. This comparison could not be conducted since Jackrabbit currently does not support transactions via the RMI interface.

5.2.5 Analysis

It turns out that Jackrabbit is faster in modifying data. The reference implementation is implemented as a central repository with a very thin client layer. This layer only delegates the client's requests to the remote repository server. This way the additional costs for network transfer are limited to transmitting the request and the response. The gathered data shows that this is clearly an advantage when modifying content. The reference implementation profits from having the central control over the repository data. It can therefore leverage caching mechanisms across transaction boundaries.

[10] `http://lucene.apache.org/`

XTC JCR plays out its advantage when processing transient items. These are managed in the client allowing for extraordinary speed. Most of the processing is done in the client allowing the system to scale very well. The comparison between Jackrabbit and XTC JCR documents the difference between a local and a distributed repository implementation. Furthermore, is shows that XTC's *appendSubtree* method is currently very slow and that Jackrabbit's caching solution is effective. Unfortunately, it turns out that the comparison is not helpful to evaluate the feasibility of the main approach which is to use a native XML database system as a back-end. The potential of a native XML database system as a back-end can hereby neither be concluded, nor is the approach invalidated.

XTC JCR is the first proof of concept implementation. If XTC JCR assumed exclusive use of the database back-end, the same cache optimizations could be employed as in the RI. Certainly, in this case, the design decision of where to put the client server gap in XTC JCR should be reconsidered. XTC JCR connects to the server transparently via RMI. Hence, integrating XTC JCR in XTC as another interface running in the same JVM would be possible without changing the overall system design. A thin remote client layer, equivalent to Jackrabbit's RMI solution, would need to be implemented.

Chapter 6

Conclusion

This thesis has given an overview of the JSR 170 specification, the foundation of *Java Content Repositories*. The specification is supported by major businesses[1] and JCR implementations are running in production environments. In short, JSR 170 is an accepted standard in the field of web application development and content management.

Although the typical setup of a JCR system is backed by a relational database, there exists an obvious analogy between JCR's hierarchic content structure and XML. The specification even exploits this fact by choosing XML as import and export format. This conceptual similarity suggests a closer analysis of how such data could be managed in a native XML database system, such as XTC.

In this work the *Java Content Repository* XTC JCR has been designed and implemented on top of XTC. The level of abstraction between the JCR data model and XML is so thin that all required JCR operations on persistent data could be mapped to DOM operations and XQuery in a straight forward fashion. Herby the approach has been proven feasible.

Modern software engineering principles have been applied to ensure high quality and maintainability of the software. The implementation has been tested and put to use in two small demo applications. Benchmarks against the Apache Jackrabbit reference implementation have shown where XTC JCR is situated in terms of performance. XTC JCR scales well and performs very well on transient operations. The current test results for persistent content modification and persistent content retrieval are disappointing. However, XTC JCR is designed for features that are currently not enabled in XTC. XTC is under constant development. These coming features will speed up XTC and XTC JCR. Then the tests should be rerun. Once the timings for modification and retrieval are under control the distributed design might even be of a great advantage.

This project has shown how elegantly the JCR data model maps to XML. It proves that the native XML database system XTC is a capable persistence back-end and shows how XML native database systems can be employed effectively

[1] Laird Popkin, 3path, Remy Maucherat, Dirk Verbeeck, ATG, Day Software, Deloitte Consulting, Hewlett-Packard, IBM, Nat Billington, Oyster Partners, SAP Portals, Software AG

in areas traditionally reigned by relational databases.

Lessons learned are:

- Network overhead must not be neglected. RMI calls are expensive. Hereby it was found that serialized results, such as Strings or XQJ result types, are generally faster than navigational access via DOM over the network.
- Test Driven Development has been very helpful. This way it was possible to control project progress and defect rate. Additionally, it helped to test the XTC system.
- Good tooling is valuable. The Netbeans IDE proved very helpful in: code editing, subversion management, unit testing, refactoring, and profiling.

Besides the constant improvement of XTC that is taking place currently, a few new issues should be addressed.

- XTC JCR full JTA support. XTC currently lacks JTA compliance. The major deficit is the lack of the *prepare* statement required for the two phase commit protocol.
- XTC JCR full versioning support. Not all methods (such as update and merge) of this feature are implemented, yet.
- Futher testing and profiling of XTC and XTC JCR under realistic conditions using JCR client applications.
- Namespace support in XTC. Instead of resolving namespaces in the XTC JCR layer the XTC itself should handle the prefix to URI mapping. For the current XTC JCR implementation this would be very beneficial in terms of performance.
- Further support for XQuery functions[2] are required for production use.
- XQJ prepared statements support. This would speed up navigational access in XTC JCR.

[2] `http://www.w3.org/TR/xpath-functions/`

List of Abbreviations

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
AST	Abstract Syntax Tree
CMS	Content Management System
CR	Content Repository
DOM	Document Object Model
EE	Enterprise Edition
ERP	Enterprise Resource Planning
FAQ	Frequently Asked Questions
FLWOR	For Let Where Order by Return
GIS	Geographic Information System
GoF	Gang of Four
IDE	Integrated Development Environment.
IO	Input Output
JAXB	Java Architecture for XML Binding
JCP	Java Community Process
JCR	Java Content Repository
JNDI	Java Naming and Directory Interface
JSF	Java Server Faces
JSR	Java Specification Request
JTA	Java Transaction API
JVM	Java Virtual Machine
OLAP	On-Line Analytical Processing
RMI	Remote Method Invocation
SAX	Simple API for XML
SQL	Standard Query Language
TAX	Tree Algebra for XML
UML	Unified Modeling Language
WCMS	Web Content Management System
WVCM	Workspace Versioning and Configuration Management
XML	Extensible Markup Language
XQJ	XQuery API for Java
XTC	XML Transaction Coordinator

Bibliography

[BBC$^+$07] Berglund, A. ; Boag, S. ; Chamberlin, D. ; Fernndez, M. F. ; Kay, M. ; Robie, J. ; Simon, J.: *XML Path Language (XPath) 2.0.* W3C Recommendation. `http://www.w3.org/TR/xpath20/`. Version: Jan 2007

[BPS00] Bray, T. ; Paoli, J. ; Sperberg-McQueen (Eds), C. M.: *"Extensible Markup Language (XML) 1.0 (2nd Edition)".* W3C Recommendation. `citeseer.ist.psu.edu/bray00extensible.html`. Version: 2000

[CBa] Chamberlin, Don ; Berglund, Anders ; al., Scott B.: *XQuery 1.0: An XML Query Language.* `http://www.w3.org/TR/xquery/`

[CD99] Clark, James ; DeRose, Steve: *XML Path Language (XPath).* `http://www.w3.org/TR/xpath/`. Version: Nov 1999

[Cle07] Clemm, Geoffrey: *JSR 147: Workspace Versioning and Configuration Management.* `http://jcp.org/en/jsr/detail?id=147`. Version: Mar 2007

[CM02] Cheung, Susan ; Matena, Vlada: *Java Transaction API (JTA).* `http://java.sun.com/javaee/technologies/jta/index.jsp`. Version: Nov 2002

[Dam07] Dambekalns, Karsten: *A Content Repository for TYPO3 5.0.* TYPO3 Developer Days 25.-29.04.2007, Dietikon / Switzerland. `http://www.typo3.org/fileadmin/teams/5.0-development/t3dd07-karsten-jcr%.pdf`. Version: Apr 2007

[DOM] *Document Object Model.* `http://www.w3.org/DOM/`

[FHK$^+$02] Fiebig, T. ; Helmer, S. ; Kanne, C.-C. ; Moerkotte, G. ; Neumann, J. ; Schiele, R. ; Westmann, T.: Anatomy of a native XML base management system. In: *The VLDB Journal* 11 (2002), Nr. 4, S. 292–314. `http://dx.doi.org/http://dx.doi.org/10.1007/s00778-002-0080-y`. – DOI http://dx.doi.org/10.1007/s00778–002–0080–y. – ISSN 1066–8888

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995

[HH07] Haustein, Michael ; Härder, Theo: An efficient infrastructure for native transactional XML processing. In: *Data Knowl. Eng.* 61 (2007), Nr. 3, S. 500–523. http://dx.doi.org/http://dx.doi.org/10.1016/j.datak.2006.06.015. – DOI http://dx.doi.org/10.1016/j.datak.2006.06.015. – ISSN 0169–023X

[Hüh08] Hühner, Stefan: *Entwicklung von Pfadoperatoren und deren Integration in eine physische XML-Algebra*, Technische Universitt Kaiserslautern, Diplomarbeit, Mar 2008

[JAKC$^+$02] Jagadish, H. V. ; Al-Khalifa, S. ; Chapman, A. ; Lakshmanan, L. V. S. ; Nierman, A. ; Paparizos, S. ; Patel, J. M. ; Srivastava, D. ; Wiwatwattana, N. ; Wu, Y. ; Yu, C.: TIMBER: A native XML database. In: *The VLDB Journal* 11 (2002), Nr. 4, S. 274–291. http://dx.doi.org/http://dx.doi.org/10.1007/s00778-002-0081-x. – DOI http://dx.doi.org/10.1007/s00778–002–0081–x. – ISSN 1066–8888

[Kaw06] Kawaguchi, Kohsuke: *JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0.* http://jcp.org/en/jsr/detail?id=222. Version: Dec 2006

[KS07] Kiselyov, Oleg ; Shan, Chung-Chieh: Lightweight Static Capabilities. In: *Electronic Notes in Theoretical Computer Science* 174 (2007), June, Nr. 7, 79–104. http://dx.doi.org/10.1016/j.entcs.2006.10.039. – DOI 10.1016/j.entcs.2006.10.039

[Mel07] Melton, Jim: *JSR 225: XQuery API for Java (XQJ) 1.0.* http://jcp.org/en/jsr/detail?id=225. Version: Oct 2007

[NN04] Nuescheler, David ; Negelmann, Björn: *E-Interview mit David Nuescheler von Day Software AG zu den Zielen und Ergebnissen der JSR 170 Initiative.* http://www.competence-site.de/cms.nsf/8AFB25D13061A6BDC1256EE1003C57E3/%$File/davidnuescheler.pdf. Version: Jul 2004

[Nue06] Nuescheler, David: *JSR 170: Content Repository for Java technology API (Release version 1.0.1).* http://jcp.org/en/jsr/detail?id=170. Version: Apr 2006

[Oes05] Oestereich, Bernd: *Analyse und Design mit UML 2.* Oldenbourg Verlag München, 2005

[OMG03] OMG: *Unified Modeling Language.* http://www.omg.org/uml/. Version: 2003

[SAX] *Simple API for XML.* http://sax.sourceforge.net/

[Sch01] Schöning, Harald: Tamino - A DBMS designed for XML. In: *Proceedings of the 17th International Conference on Data Engineering.* Washington, DC, USA : IEEE Computer Society, 2001. – ISBN 0–7695–1001–9, S. 149–154

[Sie04] Siedersleben, Johannes: *Moderne Software-Architektur.* Dpunkt Verlag, 2004. – ISBN 3898642925