|  |
| --- |
|  |
| Java GlusterFS |
| Final Deliverable |
|  |
| **CIS 4911 Senior Capstone Project**  **Section U01**  **Team Members: Maylem Gonzalez** |
| **Ian Herbig**  **Mentor: Louis Zuckerman**  **Instructor: Masoud Sadjadi**  **12/9/2014** |
|  |

|  |
| --- |
| © Copyright. Maylem Gonzalez and Ian Herbig. 2014. May be used for educational purposes without written permission but with a citation to this source.  GlusterFS is used under the GNU GPL v3.  All other works are the copyright of their respective owners. |

*Abstract*

*This document details a high-level overview of the final state of Java GlusterFS and explores the progress made throughout the project’s lifetime. In general, it covers the following topics: the scope of the project, the state of the system prior to the start of the project, the reasoning for the project’s scope, the plan of the project’s lifetime, the requirements of the system, the design of the system (system and object), and the validation of the project (testing).*

Table of Contents

[1: Introduction 5](#_Toc405933656)

[1.1: Problem Definition 5](#_Toc405933657)

[1.2: Scope 5](#_Toc405933658)

[1.3 Methodology 5](#_Toc405933659)

[1.4: Definitions, Acronyms, and Abbreviations 6](#_Toc405933660)

[1.5: Overview of Document 7](#_Toc405933661)

[2: Feasibility Study 9](#_Toc405933662)

[2.1: Description of Current System 9](#_Toc405933663)

[2.2: Alternative Solutions 10](#_Toc405933664)

[2.5: Recommendations 11](#_Toc405933665)

[3: Project Plan 12](#_Toc405933666)

[3.1: Project Organization 12](#_Toc405933667)

[3.1.1: Project Personnel Organization 12](#_Toc405933668)

[3.1.2: Hardware and Software Resources 13](#_Toc405933669)

[3.2: Identification of Tasks, Milestones, and Deliverables 14](#_Toc405933670)

[3.3 Cost 15](#_Toc405933671)

[4: Proposed System Requirements 16](#_Toc405933672)

[4.1: Functional Requirements 16](#_Toc405933673)

[4.2: Analysis of System Requirements 17](#_Toc405933674)

[4.2.2: Use Case Model 17](#_Toc405933675)

[4.2.3: Class Diagram 17](#_Toc405933676)

[4.2.4: Sequence Diagrams 18](#_Toc405933677)

[5: System Design 19](#_Toc405933678)

[5.1: Overview 19](#_Toc405933679)

[5.2: Subsystem Decomposition 21](#_Toc405933680)

[5.3: Hardware and Software Mapping 22](#_Toc405933681)

[5.4: Persistent Data Management 22](#_Toc405933682)

[5.5: Security/Privacy 22](#_Toc405933683)

[6: Detailed Design 23](#_Toc405933684)

[6.1: Overview 23](#_Toc405933685)

[6.2: Static Model 24](#_Toc405933686)

[6.3: Dynamic Model 31](#_Toc405933687)

[6.4: Code Specification 32](#_Toc405933688)

[7: System Validation 33](#_Toc405933689)

[7.1: Subsystem Tests 33](#_Toc405933690)

[7.2: System Tests 33](#_Toc405933691)

[7.3: Evaluation of Tests 33](#_Toc405933692)

[8: Glossary 34](#_Toc405933693)

[9: Appendix 35](#_Toc405933694)

[9.1: Appendix A – Project Schedule 36](#_Toc405933695)

[9.2: Appendix B – Use Cases 37](#_Toc405933696)

[9.3: Appendix C – Analysis Models 43](#_Toc405933697)

[9.5: Appendix D – Documented Class Interfaces and Constraints 49](#_Toc405933698)

[9.6: Appendix F – Documented Code for Test Drivers and Stubs 57](#_Toc405933699)

[9.7: Appendix G – Diary of Meetings 81](#_Toc405933700)

[10: References 87](#_Toc405933701)

# 1: Introduction

The Java GlusterFS is a Java implementation of the virtual file system, GlusterFS. The purpose of the project is then to bind Java to the GlusterFS. In this chapter, the GlusterFS will be explained, and a brief overview of the document will be provided.

In section 1.1, we describe the problem that the project attempts to solve. Section 1.2 then describes the scope of the project. Section 1.3 explains the methodology used. Section 1.4 provides useful definitions, acronyms and abbreviations that will be used throughout the document. Finally, section 1.5 provides a brief overview of the document as a whole.

## 1.1: Problem Definition

The Java GlusterFS project is, in its current state, an incomplete implementation of Java 7’s NIO.2 file system API backed by GlusterFS via libgfapi-jni. This project seeks to make headway towards a more complete implementation of Java’s FileSystemProvider API.

## 1.2: Scope

The Java GlusterFS binding’s extension can be described by the functionality that the system will contain after the project’s completion.

* Create directories
* Delete files and directories
* Copy files and directories
* Move files and directories
* Determine whether two files are the same
* Determine whether a directory is empty
* Retrieve the file store associated with a file
* Set and change owner/group names
* Advanced synchronous file I/O
  + Seeking & reading/writing a portion of a file

## 1.3 Methodology

This project utilizes the Agile development process, focusing on iterative development in two-week sprints and making use of constant feedback from our mentor and instructor. This approach requires that features be moved through the following stages in each sprint: (1) backlog, (2) in development, (3) testing/review, (4) acceptance, and (5) done.

This responsive approach allows us to make constant improvements to the project, with many of the features and tasks frequently moving from acceptance back to testing/review and in development before being moved to acceptance once again. In addition, all of our documentation is modified with each sprint instead of being prepared and approved ahead of time (as is done in the Waterfall model).

The constant feedback received with this approach allows us to more easily create software designs that conform to the specifications laid out by our mentor and the system requirements identified the requirements document.

The system design is represented in this documented with a series of different UML models, including

* a package diagram showing the decomposition of the system into subsystems,
* a minimal class diagram showing the relationship between the classes in the system,
* a detailed class diagram for each subsystem, and
* a state machine diagram of the main control object in each subsystem.

## 1.4: Definitions, Acronyms, and Abbreviations

* **API**: Application programming interface
* **File system**: A system used to control how data is stored and retrieved. Focuses on logical units of storage rather than physically contiguous units when applied to a computer’s method of data storage.
* **FOSS**: Free, open-source software
* **GlusterFS**: A scale-out, networked-attached storage platform
* **Horizontal scaling**: A form of scaling a system which involves adding more nodes to a system. Contrast with vertical scaling, where the size of each node in a system is larger.
* **I/O**: Input-output
* **Java Native Interface (JNI)**: A programming framework which allows Java code to call and be called by native applications and libraries written in other languages
* **libgfapi**: A library for accessing data in GlusterFS
* **libgfapi-jni**: Java Native Interface bindings for the libgfapi
* **Native applications**: Programs specific to hardware and operating system platforms
* **Network-attached storage**: file-level computer data storage connected to a computer network providing data access to a heterogeneous group of clients.
* **NIO.2**: Java 7’s enhanced file package which provides file system APIs
* **NIO**: Non-blocking I/O; also called asynchronous I/O
* **Scale-out**: see “horizontal scaling”
* **Synchronous I/O**: Input and output which blocks the progress of a program until its completion; also called blocking I/O

## 1.5: Overview of Document

This document is broken up into 10 main chapters, and each chapter is further broken up into sections and subsections. This section marks the end of the first chapter.

Chapter 2 consists of a feasibility evaluation of the Java GlusterFS project. Section 2.1 describes the state of the system prior to the project’s start, along with its known limitations. Section 2.2 details the alternative solutions considered. Section 2.3 provides an explanation of the chosen solution.

Chapter 3 describes the plan for the Java GlusterFS project. Section 3.1 introduces the project organization. Subsection 3.1.1 details the organization of the project members. Subsection 3.1.2 lists the hardware and software resources needed in completing this project. Section 3.2 concludes chapter 3 with a list of the tasks, milestones and deliverables to be completed in the duration of this project. Section 3.3 contains the analyzed cost of the project.

Chapter 4 concerns the system requirements. Section 4.1 details the functional and nonfunctional requirements determined during our analysis of the current system. Section 4.2 contains our analysis of the requirements themselves.

Chapter 5 contains the system design of the project. Section 5.1 is an overview of the chapter, identifying the architecture of the project as a whole. Section 5.2 focuses on a decomposition of the project into subsystems, describing each major subsystem. Section 5.3 maps the subsystems to hardware and software. Section 5.4 explains what, if any, persistent management concerns the project holds and our plan for managing said concerns. Section 5.5 is similar to section 5.4, but for security and privacy concerns.

Chapter 6 is similar to chapter 5, but it focuses on the detailed design of the system instead. Section 6.1 is an overview of the chapter, briefly describing the behavior and structure of each subsystem – we concern ourselves with structural and behavioral design patterns in that section. Section 6.2 provides the static model of the system, sharing a detailed description of the structure of each subsystem. Section 6.3 provides the dynamic model of the system, sharing a description of the algorithms used in each of the main control objects in each subsystem. Section 6.4 contains the code specification, describing the class interfaces contained in the appendix.

Chapter 7 is concerned with the validation of the system. Section 7.1 details the tests performed on each subsystem (code contained in the appendix). Section 7.2 details the tests performed on the system as a whole, featuring our test cases. Section 7.3 is an evaluation of the success of the various tests in the previous two sections.

Chapters 8, 9, and 10 are the metachapters in that they concern themselves with the content in the previous chapters. Chapter 8 contains the glossary of terms. Chapter 9 is the appendix. And chapter 10 contains any references we made throughout the document.

# 2: Feasibility Study

A feasibility study consists of an analysis and an evaluation of the potential of a proposed project. Within this chapter, we analyze the current system allowing us to elaborate on the reasoning for expanding its functionality. Additionally we provide high-level definitions for the user requirements which allow us to discuss alternative solutions to the problem.

## 2.1: Description of Current System

The state of the Java GlusterFS binding can be described in three forms: its functionality, the limitations on that functionality, and the system’s constraints. They are as follows:

**Functionality**:

* Able to connect to a GlusterFS volume using the NIO.2 API
* Able to perform basic synchronous file I/O
  + Reading the contents of a file all at once
  + Writing a chunk of bytes to a file all at once
* Able to handle file attributes
  + See owner/group id (as a number), size, permissions, and last modified timestamp on files and directories
  + Set file permissions
* Able to view file system/volume statistics
  + See the total, free, and usable bytes in a volume
* Directory listing (with filtering) functionality
* Able to move/rename files
* Able to watch files for changes
* Able to publish test coverage and code quality reports to SonarQube

**Limitations**:

* Unable to perform asynchronous I/O
  + Cannot seek and read/write portions of a file
* Basic error reporting and handling
* Watch system is slow and blocking, utilizing polling
* Attribute support limited
  + Owner/group identification limited to numbers, no name support
  + Cannot change owner/group

**Constraints**:

* System is written in Java
* System requires Internet access due to GlusterFS

## 2.2: Alternative Solutions

In consideration of the user requirements described in chapter 4, we must propose a series of alternatives and reduce our technical debt by analyzing them systematically for cost and feasibility concerns. The description of alternatives follows.

* **Alternative 1**: Start over and prioritize missing functionality

We could scrap the work that has been done and begin anew. In this way, we can prioritize the missing functionality from the start. This presupposes that the currently missing functionality is in some way more valuable or important than the currently present functionality, and would only make sense if the current functionality in some way impedes progress towards the missing functionality.

* **Alternative 2**: Extend existing functionality

We could keep the current system as is and simply build the new set of functionality into it. This would limit our need to cover old ground, but may run into issues if existing functionality will in some way impede progress.

* **Alternative 3**: Start over and change language

We could scrap the work that has been done and begin anew in a different language. If another language has more advanced tools to enable the integration of GlusterFS into the language, then it may streamline the process. However, it risks unknown complications that may arise. Moreover, it defeats the purpose of the entire project.

Next we must analyze each alternative for its operational, technical, schedule, and economic feasibility to enable us to recommend an appropriate solution to the problem.

## 2.5: Recommendations

Based on the alternatives analyzed in the previous section, we recommended that Alternative 2 (extending the existing functionality of the current system) be selected, as this alternative produced the best total feasibility score in our feasibility matrix (see Appendix B) and also matched the other alternatives in monetary cost (see Appendix C).

The following is a list of the major functionalities and concerns that were yet to be handled in the Java GlusterFS project:

* **Delete files**
* **Copy files**
* **Move files**
* **Create directories**
* Update watch service to use libgfchangelog (instead of polling)
* Finish attribute support
  + Owner/group names & ability to change
  + More ways to set permissions
* **Advanced synchronous file I/O**
  + **Seeking & reading/writing a portion of a file**
* Asychronous file I/O
* **Better error reporting & handling**
* Miscellaneous concerns
  + Align project versions with glusterfs (this project & libgfapi-jni)
  + Publish test coverage report to Coveralls.io

Since this list consists of some features that would each take several months to implement, or are not of immediate concern for the completion of the project, we further recommended that the functionalities bolded above be prioritized in the four months. These recommended functionalities would require the least amount of external aid from community members, and also represent some of the most important features that any user would expect to be implemented when using Java GlusterFS.

# 3: Project Plan

This chapter focuses on the details of project management. We specifically focus on the project organization and the identification of tasks, milestones, and deliverables. Within, we present an initial timetable for the project and allot time to each aspect of the project.

## 3.1: Project Organization

The organization of the Java GlusterFS project for the remainder of this semester is a loose organization with a fair amount of redundancy in roles between the two project members. These roles will be elaborated on in some minor detail. In addition, this section will detail the various hardware and software resources that will be used throughout the project’s lifespan.

### 3.1.1: Project Personnel Organization

The two individuals working as a team within this project share a measure of redundancy in their roles due to the relatively small size of the team and the nature of the project itself. However, there remains a measure of specialization, so the roles will be described as follows:

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Description** |
| **Ian Herbig** | Project Manager/Software Engineer/Tester/Document Editor | Serves as the project manager for the other team member. Implements functional requirements. Performs unit and integration tests. Manages the documentation. |
| **Maylem Gonzalez** | Project Manager/Software Engineer/Tester/Minute-Taker | Serves as the project manager for the other team member. Implements functional requirements. Performs unit and integration tests. Takes minutes during meetings. |

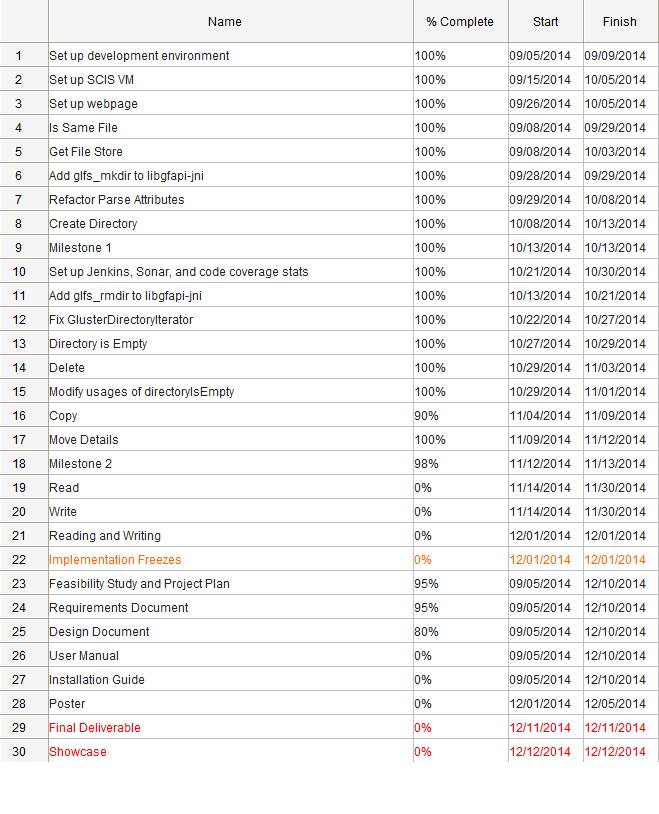
### 3.1.2: Hardware and Software Resources

This section details the various hardware and software needs of the project.

|  |  |
| --- | --- |
| ***Hardware*** | ***Software*** |
| Minimum 1GB RAM (recommended 2GB) | OpenJDK 7 |
| Internet access | Oracle JDK 7 |
| Minimum 20GB storage | Apache Maven v3.0 |
| Minimum 1GHz processor (recommended 1.6GHz) | IntelliJ IDEA |
| 64-bit processor | Lombok plugin for IntelliJ IDEA |
| Portability recommended | Vagrant |
| Minimum 1024x768 display | Software contained in the glusterfs-common package |
|  | Git |
|  | JUnit |
|  | A 64-bit Linux operating system |
|  | Gantter.com |
|  | Trello |
|  | SonarQube |
|  | Jenkins |
|  | Atlassian Clover |

## 3.2: Identification of Tasks, Milestones, and Deliverables

The following table is an incomplete schedule subject to revision.



## 3.3 Cost

|  |  |
| --- | --- |
| Project Component | Monetary Cost |
| Hardware | $0 |
| Software | $0 |
| Personnel | $0 |
| Training | $0 |
| Testing | $0 |
| Total | $0 |

One of the benefits of being a FOSS project is the “free” aspect of it.

# 4: Proposed System Requirements

This chapter details the tentative functional and non-functional requirements attained by analyzing the high-level user requirements. Section 4.1 describes the functionality on a high level by enumerating functionality the system shall have and the associated non-functional requirements for each element of that set as required.

Following this, we analyze these system requirements in section 4.2 with an emphasis on generating scenarios from which we can draw specific use cases to be implemented. Models and diagrams follow naturally.

## 4.1: Functional Requirements

* *The system shall allow a user to delete a file from a GlusterFS volume*
* *The system shall allow a user to copy a file within a GlusterFS volume*
* *The system shall allow a user to move a file within a GlusterFS volume*
* *The system shall allow a user to read a portion of a file*
* *The system shall allow a user to write to a portion of a file*
* *The system shall allow a user to create a directory*
* *The system shall allow a user to determine whether two files are the same*
* *The system shall allow a user to retrieve the file store that a file is associated with*

The above functional requirements do not have non-functional requirements that could be considered under the umbrella of usability, except those specified by the Java 7 NIO.2 API. They must work in 90% of the cases in which they are used, barring exceptional circumstances. And copying, moving, reading, and writing, are variables in terms of performance as they are dependent on the size of the file. A “small” file should not take a noticeable amount of time.

All other non-functional requirements of the system are specified by the non-functional concerns of the depending systems (libgfapi and glusterfs).

## 4.2: Analysis of System Requirements

This section contains an explanation of the diagrams displayed in the appendices and the scenarios that helped build them.

### 4.2.2: Use Case Model

The diagram in appendix C provides a limited overview of the use cases we have determined. We specify two types of users in the diagram: the User is someone who wishes to utilize GlusterFS from within a Java program; the System is involved in the calling of private methods involved in the execution of system methods.

Some of the use cases interact with some other functionality that we describe, but as this process is iterative, the diagram and use cases will likely be expanded upon in the future as progress on the system is made and insights are had into what may be necessary for the completion of future milestones.

### 4.2.3: Class Diagram

The third diagram visible in appendix D (Figure 1) is the minimal class diagram for the Java GlusterFS system, while the second class diagram in appendix D (Figure 2) shows the parents of each of the classes. Though they are not broken up as such, almost every class in the class diagrams can be seen to make up a subsystem in itself. Exception to this are classes like the GlusterDirectoryStream and GlusterDirectoryIterator which both make up the subsystem responsible for parsing the contents of a directory.

The core classes, GlusterFileSystemProvider, GlusterFileSystem, and GlusterPath, are the classes in which most user-driven logic is performed and are the focus of our work on the project (excluding our extension of the libgfapi-jni project, which is merely creating JNI bindings to the libgfapi project). Users do not typically see the aforementioned classes directly, preferring instead to go through the NIO.2 Files API which enables generic typing.

### 4.2.4: Sequence Diagrams

The sequence diagrams contained within appendix D are based on our work on the current system and reflect a general idea for the flow of control throughout the system. Generally, when a function is user-driven, the sequence diagram will reflect that the user goes through the Files class rather than handling the Java GlusterFS library itself. Pay special attention to notes within the diagrams as the notes indicate where functionality may be lacking, or assumptions that are made.

For example, in the isDirectoryEmpty sequence diagram, a note is made that the directory entries parsed by the GlusterDirectoryIterator are checked against the special directory entries of “.” and “..” which represent “this” and “parent” respectively, and discarded if they match. This functionality is something we aim to implement, and the diagram reflects the current state of the function.

# 5: System Design

In this chapter, we will focus on the high-level design of the system by concerning ourselves primarily with the architectural design of the system and the decomposition of the system into subsystems. Once the system has been broken down into subsystems, we will then map those to hardware and software.

The final points for this chapter will be about persistent data management and security/privacy concerns. More specifically, those sections will detail why this project is unconcerned with privacy and security, the reasoning for that, and the extent to which our project manages persistent data.

## 5.1: Overview

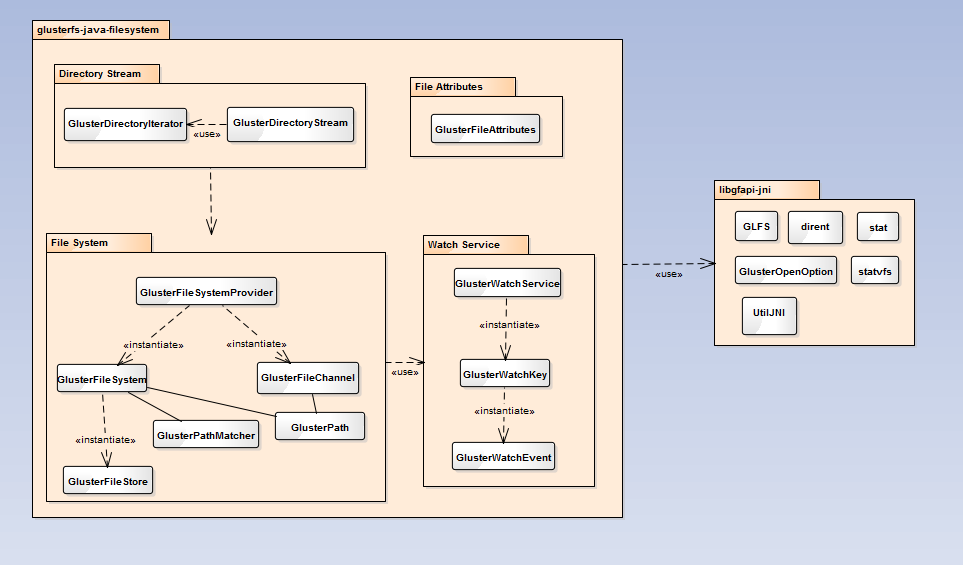
The Java GlusterFS system is broken down into four subsystems: Directory Stream, File Attributes, File System, and Watch Service. The package diagram (Figure 2.1.1) below depicts the package decomposition of the system.

The classes concerned with providing a watch service (e.g. to watch a directory for changes) reside in the Watch Service subsystem. The GlusterFileAttributes class, which encapsulates a number of file attributes, is located in the File Attributes subsystem. The classes that deal with providing a means for iterating through available directories are found in the Directory Stream subsystem.

Finally, the classes that concern themselves with file system tasks (i.e. manipulating files in a file system) can be found in the File System subsystem. It is in this subsystem that the logic for the project requirements (e.g. creating directories, deleting files, copying files, etc.) will be implemented. It is worth noting that the gluster-java-filesystem package makes use of the libgfapi-jni package, since the latter houses the Java Native Interface bindings for the Gluster filesystem’s client API, libgfapi. While modifications to the libgfapi-jni package will need to be made in this project, since the libgfapi-jni package serves merely as a library, and since most of this code is auto-generated, the focus of this chapter will remain with the glusterfs-java-filesystem package.

Two of the architectural patterns used in relating the requirements of the Java GlusterFS project to the system’s composition were the Object-Oriented and Service Provider Interface patterns. The Object-Oriented pattern focuses on the division of responsibilities in a system to show the relationships and interactions between classes and objects. These reusable objects contain the data relevant to the desired behavior of the object. This architectural pattern can be seen throughout the Java GlusterFS system; for example, The GlusterFileSystem class contains all the data relevant to a Gluster file system object, including a Gluster file system provider, host name, and other pertinent data. The Service Provider Interface is a Java pattern that allows for framework extension (e.g. the implementation or extension of an API). Evidence of the pattern is also easily found throughout the Java GlusterFS project, since the project implements Java’s FileSystemProvider API and extends several classes from the java.nio.file package.

Figure 2.1.1: Package Diagram



## 5.2: Subsystem Decomposition

**Directory Stream**: The Directory Stream subsystem consists of two classes, GlusterDirectoryIterator and GlusterDirectoryStream. These classes house the logic that allows for the parsing of available directories, which in turn allows us to fulfill the requirement of determining whether a directory is empty. This requirement is crucial in fulfilling the system requirements of deleting directories (which can only be done if a directory is empty), moving directories, and copying directories.

**File Attributes**: The File Attributes subsystem consists of one class, GlusterFileAttributes, which contains a list of file attributes that a user may use for file creation and file manipulation. It is worth noting that this subsystem has no association to the other subsystems. This is due to the fact that the contents of this class provides optional file attributes for the user, but these attributes are not enforced elsewhere in the system.

**File System**: The File System subsystem houses the logic for all of the system requirements detailed in this project. The logic for the read and write requirements is found in the GlusterFileChannel class, and the main logic for the rest of the requirements can be found in the GlusterFileSystemProvider class.

**Watch Service**: The Watch Service subsystem contains the logic that allows a user to watch files and directories for changes. However, the logic in this subsystem was not tied to any of the system requirements completed during this course. Future versions of this project may utilize this subsystem, as the watch service still needs to be fully implemented.

## 5.3: Hardware and Software Mapping

It is required that the Java GlusterFS subsystems be developed and used on a computer with a 64-bit architecture due to the Gluster file system needing a 64-bit Linux operating system. In addition, the subsystems will not be guaranteed to produce correct output if one of the following operating systems is not used: CentOS (5.1 or greater), RHEL (5.1 or greater), Ubuntu (8.04 or greater) and Fedora 11. An underlying POSIX compliant disk file system is also required in order for the File Attributes subsystem as well as the File System subsystem to set and modify file attributes properly. All of the subsystems are also mapped to a storage machine or VM (virtual machine) with a 64-bit processor, 8 GB minimum of storage space, and 1 GB of memory.

## 5.4: Persistent Data Management

The Java GlusterFS project does not contain persistent data elements that need to be stored for the system to properly function; the system instead serves as a tool for the manipulation and creation of persistent data in a file system. Since a discussion of how the system can be used to create persistent data is beyond the scope of this project, persistent data management will not be considered in this document.

## 5.5: Security/Privacy

The Gluster file system does not currently support encryption or any other means of enforcing security and privacy. While the Java GlusterFS project allows a user to set and change the permissions of a file or directory, since the Gluster file system itself cannot keep someone connected to the Gluster server from accessing any of the data in the volume, the setting of such permissions does not truly serve to provide any real security.

We discussed enforcing security within our Java GlusterFS library (e.g. forcing permissions to be checked every time a file or directory is to be deleted or manipulated in some way), but this would force users to conform to our security scheme in order to utilize our library. Since this may not be desirable for the user, and since concerns about external systems that may be used to protect data in a Gluster server are beyond the scope of this project, this document will not concern itself with security or privacy.

# 6: Detailed Design

This chapter on detailed design focuses on the specific modeling of the system. It includes class diagrams, state diagrams, the description of each, and the description of the class interfaces found in Appendix C.

## 6.1: Overview

In the previous chapter, the system was broken down into four main subsystems: directory stream, file system, file attributes, and watch service. Of these, the file attributes subsystem is more or less a utility subsystem serving to define file attributes in case the user wishes to use them; the other subsystems generally do not interact with the file attributes subsystem except in the cases related to specific file manipulation. This is related to the discussion concerning security/privacy in the previous chapter.

The directory stream subsystem enables the user to parse through the entries in a directory, and so is reliant on the file system subsystem. The subsystem itself is structured very simply, with the GlusterDirectoryStream creating and using a GlusterDirectoryIterator: the stream serving to encapsulate the iterator and the data it retrieves.

The more complicated subsystems are the watch service and file system subsystems. The watch service consists of a three-tier structure: the service creates a key which creates an event. The service is initiated by the user, but the file system subsystem ties into the watch service in the obvious way: that is, the watch service watches files or directories for changes, and the file system subsystem is managing those changes.

The file system subsystem is the most complicated of the subsystems featuring twice the number of classes as the watch service. This subsystem can be thought of as a tiered system, with additional components utilized throughout. The GlusterFileSystemProvider creates the GlusterFileSystem which creates the GlusterFileStore: this mimics the general structure of Linux-based file systems. In addition to the GlusterFileSystem, the GlusterFileSystemProvider creates a GlusterFileChannel utilized in the file I/O processes.

This structure enables the GlusterFileSystemProvider to serve as the primary entry-point for user-driven logic into the subsystem, delegating out tasks when it needs to. Due to this structure, users of the system generally need not concern themselves with the details of the subsystem (though, they can if they wish). In fact, due to the nature of the system as an implementation of the Java 7 NIO.2 Files API, users need not concern themselves with the structure of the system at all so long as they understand how to use the Files API which abstracts the implementation away.

## 6.2: Static Model

The class structure and associations for Java GlusterFS are specified by the Java NIO.2 file system API. This is depicted in Figure 6.2.1, which shows the minimal class diagram of Java GlusterFS with each class expressing its parent class or interface. As can be seen, every class in Java GlusterFS extends a class or implements an interface found in Java 7’s file system API.

Figure 6.2.1: Minimal Class Diagram with Parents

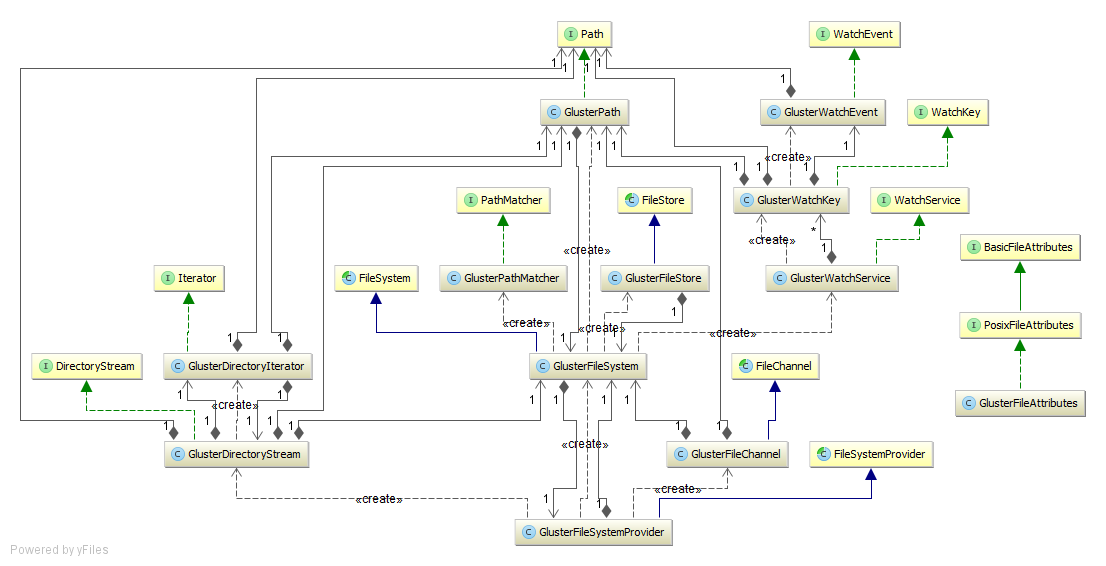
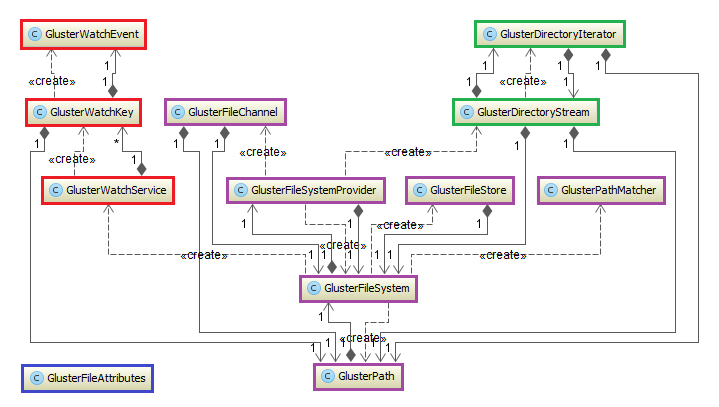


Figure 6.2.2 groups those classes in the same subsystem together by color and shows the associations between these classes. The following legend shows the subsystem associated with each color.

* Red: Watch Service
* Blue: File Attributes
* Green: Directory Stream
* Purple: File System

As can be seen, GlusterFileAttributes is not associated with any of the other classes in the system for the same reason that the subsystem that it belongs to, GlusterFileAttributes, is not. The logic housed in this subsystem concerning file attributes is made available for the user, but is not enforced in any of the other subsystems due to security considerations; mainly, we do not want to force a user into the potential situation of being unable to gain access to certain files in the Gluster volume because of file permissions, when the Gluster file system itself allows all connected users to access all files in the volume regardless of file attributes. We decided that this would be an unexpected behavior of our library for those users who have used GlusterFS in the past and are accustomed to being able to access all files in the volume.

Figure 6.2.2: Minimal Class Diagram with Subsystems Highlighted

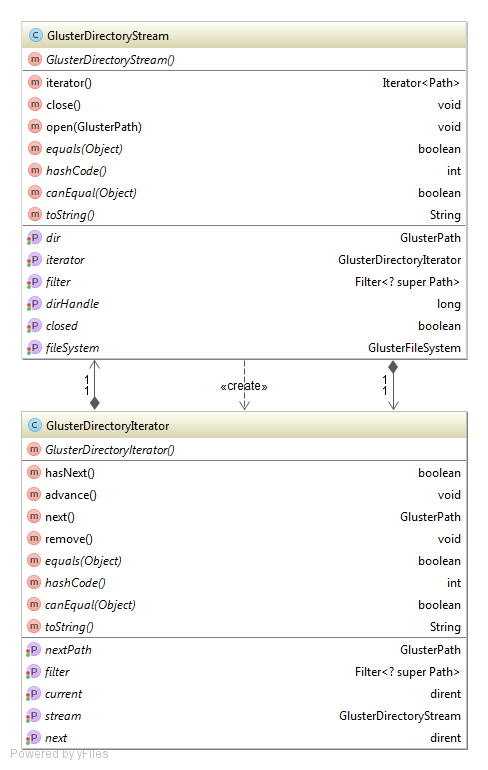


What follows will be an in-depth discussion of the structure of each subsystem and the design patterns evident in these systems.

**Directory Stream**:

The class diagram of the Directory Stream subsystem is depicted in Figure 6.2.3, with the constructors, methods, and class properties shown for each of the classes. A design pattern used in this subsystem is the ***Iterator*** ***pattern***, which allows a user to access the elements of an aggregate object (in this case, the file directories) without exposing to the user any of the underlying logic to do so. The public GlusterDirectoryStream class creates and returns a GlusterDirectoryIterator, which allows the user to easily parse through directories in a path using methods like hasNext() and next().

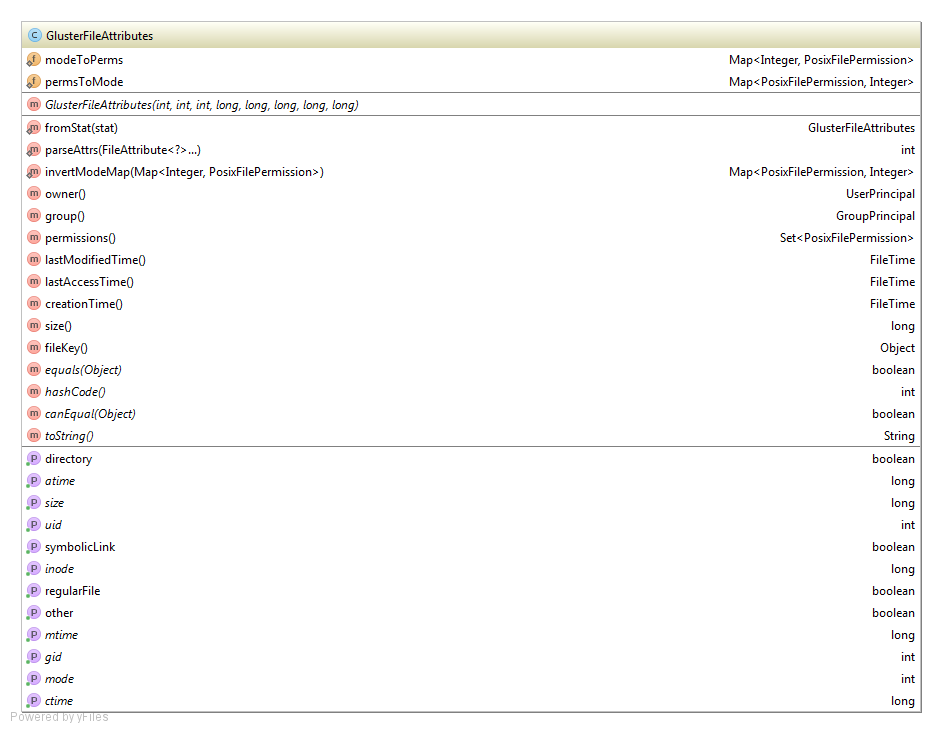
Figure 6.2.3: Directory Stream Subsystem



**File Attributes**:

The class diagram for the File Attributes subsystem is shown in Figure 3.2.4. This class holds the methods for parsing file attributes and retrieving information about a file such as the file’s creation time, the file’s owner and the permissions on the file. In addition, this class maps file modes in octal notation to a Posix file permission, as well as Posix file permissions to an octal file mode. Posix file permissions are broken up into execute, write, and read for each of the three permission components (owner, group, and others), as is common in Unix permissions.

Figure 6.2.4: File Attributes Subsystem



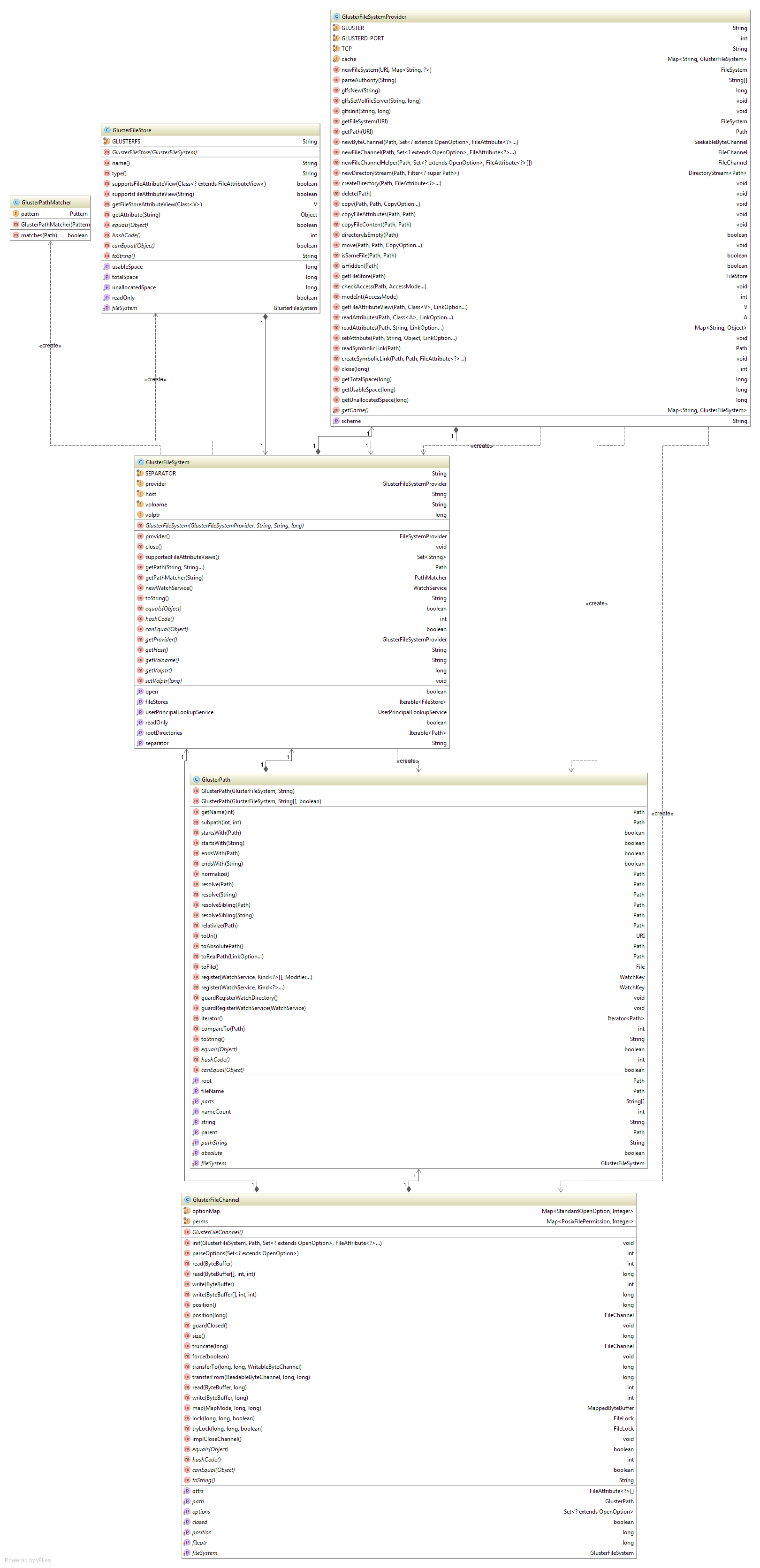
**File System**:

The class diagram for the File System subsystem can be seen in Figure 3.2.5. This subsystem is by far the most complex of the subsystem, and is the central logic hub of the Java GlusterFS project. This subsystem is tasked with the creation of the necessary objects for a file system (file system, file channel, file store, and file path objects) and houses most of the methods used in common file system tasks. Of particular importance to our project was the FileSystemProvider class, which houses the majority of the logic that was needed in implementing the requirements outlined for this project. Important methods in this class include the user accessible methods for deleting, copying, and moving files.

The File System subsystem includes several design patterns. The ***Singleton pattern*** can be seen with the GlusterFileSystem object, of which there can only ever be one instance in the class. This is useful in our project, since there can only ever be one type of file system in a Gluster volume. Another pattern used in this system is the ***Factory pattern***. This pattern consists of an interface that provides for the creation of objects. For example, in the Java GlusterFS library, the GlusterFileSystemProvider factory creates GlusterFileSystem objects from user-provided URIs.

Another design pattern that is prevalent in this subsystem is the ***Fluent Interface pattern***. The goal of this pattern is to make more readable code, and it is usually implemented through method cascading. The Java GlusterFS project was made with this pattern in mind, and examples of it can be found throughout both the project and its libgfapi-jni library. An example of this pattern in use can be found in the GlusterFileSystemProvider class with code that looks as follows: “path.getFileSystem().getFileStores().iterator().next();” Of course, the main drawback to this approach is the fact that it is much more difficult to debug the code when several methods are chained into one line, but, on the other hand, it is much easier for developers to write and read the code.

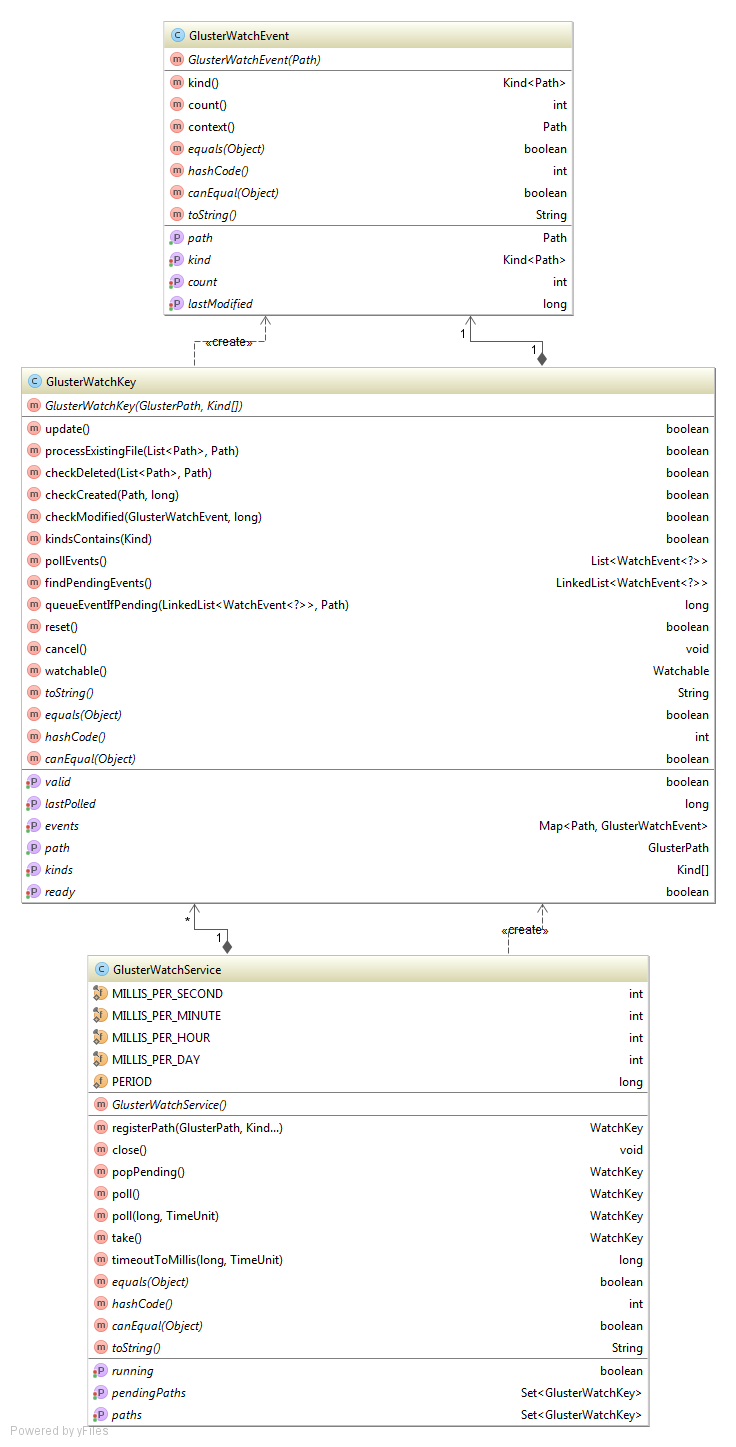
**Figure 6.2.5: File System Subsystem**



**Watch Service**:

The class diagram for the Watch Service subsystem can be seen in Figure 3.2.6. This three-tiered subsystem uses polling to notify the user when a change has been made to an object, in this case a directory. A GlusterWatchService returns a GlusterWatchKey object that represents the registration of the watchable object to the watch service. The GlusterWatchKey in turn returns GlusterWatchEvent objects when polling occurs to represent the type of events that occurred to the directory, as well as the number of times that this event occurred.

Figure 6.2.6: Watch Service Subsystem

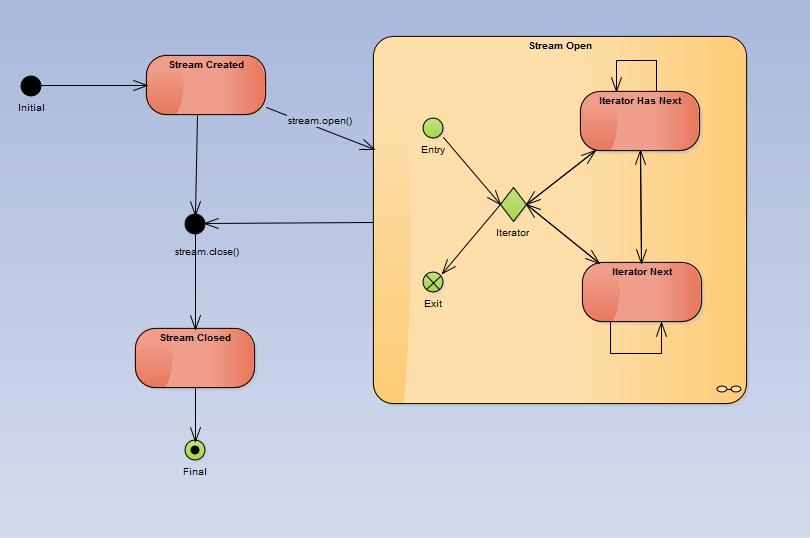


## 6.3 Dynamic Model

In the Directory Stream subsystem, the main control object is the GlusterDirectoryStream object. Figure 6.3.1 depicts a state machine diagram of this object. When the user creates a directory stream, the state of the object is S*tream Created*, and the user can at this point choose to either call *stream.open()* or close the stream without doing anything using *stream.close()*.

If *stream.open()* is called, the object enters the entry point of the *Stream Open* state. In this state, a user can utilize the steam's iterator to and call the iterator's *hasNext()* and *next()* as often as the user desires (or choose not to call them at all) before exiting the state. At this point, the stream can be closed using *stream.close()*, which is the last possible state for the directory stream object.

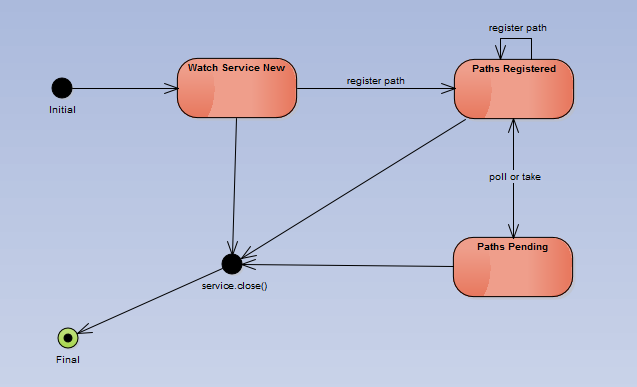
Figure 6.3.1: Directory Stream Subsystem State Machine



The main control object of the Watch Service subsystem is the GlusterWatchService object. Figure 6.3.1 depicts a state machine diagram of this object. When a GlusterWatchService is created, the state of the object is *Watch Service New*, and the user can either choose to register a path with the watch service by calling *registerPath()*, or close the service without using it using *service.close()*. If the user registers a path, the state changes to *Paths Registered*. The user can remain in this state by continuing to register other paths to the watch service.

From this state, the user can either call *service.close()* to move to the *Stream Closed* state, or call *poll()* or *take()* on the watch service to move to the *Paths Pending* state. In this state, the user can choose to go back to the *Paths Registered* state to register more paths or continue to poll the list of pending paths. Once done, the user can call *service.close()*, moving the object to the *Stream Closed* state, the last state of the object’s state machine diagram.

Figure 6.3.2: Watch Service Subsystem State Machine



The main control object in the File Attributes subsystem, the GlusterFileAttributes object, is used to store a file’s information and does not have any change in state after being created. Lacking any algorithm that changes this object’s state, a state machine diagram could not be made for this subsystem.

In addition, the main control object for the File System subsystem (GlusterFileSystemProvider object) does contain a state, but being a factory object it does not contain a state in the traditional sense; it is immutable and as such has exactly one state that does not change. The classes in the File System subsystem concern themselves with the details of manipulating the Gluster volume (creating files, copying directories, etc), and therefore also do not house an algorithm that changes the state of the GlusterFileSystemProvider object. For this reason, a state machine diagram could not be made for the File System subsystem.

## 6.4 Code Specification

The main control objects of the four subsystems in Java GlusterFS are the GlusterFileAttributes, GlusterDirectoryStream, GlusterFileSystemProvider, and GlusterWatchService objects. What follows is a discussion about each of these classes and their constraints.

**GlusterFileAttributes**:

The GlusterFileAttributes class depicts a snapshot of a file’s state, and as such the methods in the class largely concern themselves with retrieving the data about the file. As such, the only pre-condition that could be considered to exist in this class is that the GlusterFileAttributes object was correctly created, and no post-conditions or invariants exist in this class.

The class implements PosixFileAttributes and contains maps to match POSIX octal file modes to PosixFilePermissions, and PosixFilePermissions to octal file modes. The object constructor takes 3 ints and 5 longs, consisting of a file’s mode, owner, group, size, last access time, last status change time, last modification time and the file’s inode. The *fromStat()* method takes a stat object, which contains a file’s attributes, and returns a new GlusterFileAttributes object using this constructor.

The *parseAttrs()* method takes a set of FileAttribute objects and returns an int representing the octal mode value of the file’s permissions. The *owner()* method is called on a GlusterFileAttributes object and returns the owner of the file as a UserPrincipal object. The *group()* method returns the group owner of the file as a GroupPrincipal object. The *permissions()* method returns the set of PosixFilePermissions of a GlusterFileAttributes object.

The *lastModifiedTime()*, *lastAccessTime()*, and *creationTime()* methods return FileTime objects containing a file’s last modified time, last access time, and last status change time, respectively. The *isRegularFile()*, *isDirectory()*, *isSymbolicLink(),* and *isOther()* methods return a boolean value representing whether the file is a file, a directory, a symbolic link, or some other object (like a hard link), respectively. The *size* method returns the file of the size, and the *filekey()* method returns the inode of the file.

**GlusterDirectoryStream**:

The constructor in the GlusterDirectoryStream class does not take any parameters. However, to actually open a directory stream, using the class’ *open()* method, the directory that the given path represents must exist. If this pre-condition is violated, then a directory stream will fail to be opened. As such, an invariant for all instances of this class is the continued existence of the directory passed to the opened directory stream, as well as the continued existence of the Gluster volume. A post-condition of this method is a directory stream tied to the provided directory.

The *iterator()* method returns a GlusterDirectoryIterator object that can be used to parse the directories in the opened directory stream. As such, an open directory stream is a pre-condition for this method, and the method’s post-condition consists of the user being able to call the iterator’s methods (like *hasNext()* and *next()*). Finally, the *close()* method closes a directory stream. A pre-condition for this method is that the directory stream must have been opened, and the post-condition is that the directory stream is no longer associated to the directory previously provided.

**GlusterFileSystemProvider**:

Like the GlusterFileAttributes class, there is no algorithm that alters the state of the GlusterFileSystemProvider object. As such the only class-wide pre-condition is the fact that the Gluster volume being manipulated must exist, and the only invariant is that the Gluster volume must continue to exist until the provider object is disassociated from it.

This class contains several methods to associate the GlusterFileSystemProvider object with the Gluster volume being used, such as *glfsNew()* and *glfsInit()*, and to disassociate the provider object with the volume (using the *close* method). All other methods independently manipulate data on the Gluster volume or provide the user with information about the volume.

The *createDirectory()* method takes a Path object and an optional set of FileAttribute objects. A pre-condition for this method is that the path provided does not exist and that the parent of the path exists (so, for example, if the path provided is /B/A, B must already exist in order for A to be created). The post-condition of this method is a new directory created in the specified path in the Gluster volume, either with default permissions or with the provided permissions.

The *delete()* method takes a Path object. A pre-condition for this method is that the Path provided exists, and that, if the path leads to a directory, the directory be empty. If these hold true, then the post-condition of calling this method is the deletion of the specified file or directory from the Gluster volume.

The *copy()* method takes a source Path object, a target Path object, and any number of CopyOption objects. A pre-condition for this method is that the two paths provided be absolute, and not relative, paths, in the Gluster volume, and that the source path exists. Additionally, if the target path already exists, then the REPLACE\_EXISTING copy option must have been provided, and if the target path is a directory, then this directory must be empty. Finally, the ATOMIC\_MOVE copy option must not have been specified, as it is not supported in this library. The post-condition of this method is that the source file or directory will be copied to the target path in the Gluster volume.

The *directoryIsEmpty()* method takes a Path object and returns a boolean representing whether the directory specified is empty. The only pre-condition for this method is that the file specified by the path exists and that it is a directory. The post-condition for this method is a boolean value of true being returned if the directory is empty, or false if it is not.

The *move()* method takes a source Path object, a target Path object, and any number of CopyOption objects. Pre-conditions for this method consist of the source and target paths both being absolute paths in the Gluster volume, and the source path must exist. In addition, if the target path already exists, then the REPLACE\_EXISTING copy option must have been provided, and if the target path is a directory, then this directory must be empty. As with the *copy()* method, ATOMIC\_MOVE is not supported, so this must not be a specified copy option. Finally, the file systems for both paths must the same. The post-condition of this method is the source file or directory being either renamed (if the source and target paths lead to the same directory and the file or directory names differ) or deleted from the source path and moved to the target location.

The *isSameFile()* method takes two Path objects and returns a boolean representing whether the two files are the same. If the two paths reside in different file systems, then the method returns false. If the two paths are identical, or if the inodes of the two paths are the same, then the method returns true. The post-condition is either true or false being returned depending on the paths specified. The *getFileStore()* method takes a Path object and returns a FileStore object. The only pre-condition for this method is that the file specified by the path exists. The post-condition of this method is a FileStore object associated with the file specified being returned to the user.

**GlusterWatchService**:

The constructor in the GlusterWatchService class does not take any parameters. An invariant for all instances of this class is the continued existence of the Gluster volume; each GlusterWatchService instance keeps track of the paths registered to the watch service, and all of these paths must reside within a Gluster volume.

The *registerPath()* method returns a new GlusterWatchKey object to represent the registration of the path and adds the watch key object to a collection. If the path specified has already been registered, then the existing watch key is found and returned. A pre-condition of this method is that the watch service object must not have been closed. The *close()* method closes the watch service if it had not previously been closed; otherwise, the method returns without doing anything.

The *poll()* method retrieves and removes the next watch key from the list of registered watch keys. A pre-condition for this method is that the watch service must still be running (so *close()* must not have been called). The post-condition for this method is the user receiving the next watch key object, or if no other watch key objects are to be found, then null is returned.

There is a second *poll()* method that works in the same manner but takes a long value representing the timeout and a TimeUnit value. The method has the same pre-conditions, but this time it continues to poll for the duration of time specified in the arguments before the method returns. This is done so that if a WatchKey is expected to be added, the user can specify a reasonable time for the method to wait before returning.

Finally, the *take()* method works in the same manner as the *poll()* methods, except that the user cannot specify the amount of time the method should wait before returning; instead, *take()* waits until the next watch key is added if none can be found. As before, the pre-condition for *take()* is that the watch service be running.

# 7: System Validation

In this chapter, we will describe the tests we performed to validate the system. The tests take two forms: unit and integration. The unit tests rely heavily on the JUnit, Mockito, and PowerMock testing frameworks, while the integration tests are less involved.

## 7.1: Subsystem Tests

The unit tests in Appendix F rely heavily on the JUnit, Mockito, and PowerMock testing frameworks. JUnit enabled us to format the tests into test suites and test cases. Each test suite exists for an individual class, while each test case exists for a method of that class. JUnit also provides the tools to “expect” certain output, either asserting that some result is as we expect it to be, or declaring that a given method will throw an exception under the conditions we stub.

Stubbing is done through Mockito and PowerMock. Rather than actually creating stubbed classes and methods in those classes which return the values we want, Mockito enables us to create “mock” objects which behave only in the manner we specify. Once a mock object is created, we can inject the expected behaviors into it, so the actual classes and methods are not under scrutiny, only the code within the specific method to be tested.

Mockito is limited, however, to mocking standard objects. It is unable to mock static or final classes or methods. This project makes ample use of such tools, so we needed some way to be able to ensure that only the method under test would actually be run. That is where PowerMock comes in. PowerMock enables us to mock static and final objects and methods.

For instance, in many of the methods we call *Files.exists()* to test if a specific path exists before logic continues. Under the standard Mockito, we would have no choice but to ensure that a given condition is externally true (a path exist or does not exist depending on the test case) in order to run our test case. PowerMock enables us to simply call *mockStatic(Files.class)*, and then we can treat the Files class as if it were a mock object and inject any methods in.

Additionally, Mockito and PowerMock enable the verification of the mocked objects. So not only can we say *doReturn(10).when(foo).bar()*, which would make some mock object *foo* return 10 when it called some method *bar()*, we can verify that the method call was actually made by saying *verify(foo).bar()*. This enables us to ensure that the method under test is performing the expected method calls as well as ensuring that the method calls have stubbed values.

Throughout the project, we wrote 30 unit tests which raised the overall test code coverage percentage from 82.9% to 84.2%.

## 7.2: System Tests

The integration testing we performed is less involved. Due to the simplicity of code which the NIO.2 API enables, we were able to succinctly test the functionality we added in a single class. The code is visible in Appendix F, but will be summed up here. As security is not something that the Java GlusterFS project concerns itself with, there are no test cases which test the security of the system.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test Case ID | Purpose | Test Setup Environment | Test Inputs | | Expected Outputs |
| Create Directory | Create Directory Test 1 | Test if the directory is properly created. | Ensure the path to the directory does not already contain any files. | An absolute path to the directory. | | None visual in code, but the directory should exist after the test. |
|  | Create Directory Test 2 | Test if the directory creation properly fails when an object already exists at the specified path. | Ensure a file exists at the specified path. | An absolute path to the existing file. | A message to the console indicating failure and a FileAlreadyExistsException is thrown. | |
|  | Create Directory Test 3 | Test if the directory creation properly fails when there is a problem with the connection to the gluster volume. | Ensure no gluster volume is accessible at the specified IP address. | An absolute path to a file. | | A message to the console indicating failure and an IOException is thrown. |
| Copy | Copy Test 1 | Test if a file is properly copied; both attributes and content. | Ensure the file to be copied exists. | The absolute path to the file to be copied. The absolute path to the target location. The appropriate flags to specify copying of file content and attributes. | | A message to console indicating success. The file should also be properly copied as indicated by a diff on the original and target files and a stat on the two files to compare attributes. |
|  | Copy Test 2 | Test if the copying process properly fails when there is a problem with the connection to the gluster volume. | Ensure no gluster volume is accessible at the specified IP address. | Same as above. | | A message to the console indicating failure and an IOException is thrown. |
| Get File Store | Get File Store Test 1 | Test if getFileStore properly retrieves the file store of a file on a gluster volume. | Ensure the file to retrieve the file store for exists. | The absolute path to the file. | | A message to the console detailing the file store. Should say gluster in it. |
|  | Get File Store Test 2 | Test if getFileStore properly fails when the file does not exist. | Ensure the file to retrieve the file store for does not exist. | The absolute path to the fake file. | | A message to the console indicating failure and an IOException is thrown. |
| Is Same File | Is Same File Test 1 | Test if isSameFile returns true when passed the same path twice. | Nothing unique to this test. | The absolute path to a single file, file may or may not exist. | | true should be returned. |
|  | Is Same File Test 2 | Test if isSameFile returns false when passed a file and a copy of a file. | Ensure both files passed as parameters exist. | The absolute path to two files: a file, and its copy. | | false should be returned. |
|  | Is Same File Test 3 | Test if isSameFile returns true when passed a file and a hard link to the file. | Ensure both files passed as parameters exist. | The absolute path to two files: a file, and a hard link to the file. | | true should be returned. |
|  | Is Same File Test 4 | Test if isSameFile returns true when passed a file and a symbolic link to the file. | Ensure both files passed as parameters exist. | The absolute path to two files: a file, and a symbolic link to the file. | | true should be returned. |
| Delete | Delete Test 1 | Test if delete can delete an empty directory. | Ensure the directory exists and is empty. | The absolute path to the empty directory. | | A message to the console should be printed indicating success and the directory should be deleted. |
|  | Delete Test 2 | Test if delete properly fails when trying to delete a file that does not exist. | Ensure the file being deleted does not exist. | The absolute path to a fake file. | | A message indicating proper failure and an IOException is thrown. |
|  | Delete Test 3 | Test if delete can delete an ordinary file. | Ensure the file being deleted exists. | The absolute path to the file. | | A message indicating success should be printed to the console and the file should be deleted. |
|  | Delete Test 4 | Test if delete properly fails when trying to delete a non-empty directory. | Ensure the directory being deleted exists and has at least one file in it. | The absolute path to the directory. | | A message indicating proper failure and an IOException is thrown. |
|  | Delete Test 5 | Test if delete can delete symbolic links. | Ensure the symbolic link being deleted exists. | The absolute path to the symbolic link. | | A message indicating success should be printed to the console and the link should be deleted. |
|  | Delete Test 6 | Test if delete can delete hard links. | Ensure the hard link being deleted exists. | The absolute path to the hard link. | | A message indicating success should be printed to the console and the link should be deleted. |

## 7.3: Evaluation of Tests

The following table examines the tests for their success. Success for unit tests is determined by JUnit upon running of the test. As a prerequisite for inclusion into the larger project, the unit tests must all pass, and so they are excluded from the following table. The integration tests are more involved, and so their passing status will be mentioned as well as any concerns raised during testing.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test ID | Success Status | Concerns |
| Create Directory | Create Directory Test 1 | Passed | N/A |
|  | Create Directory Test 2 | Passed | N/A |
|  | Create Directory Test 3 | Passed | N/A |
| Copy | Copy Test 1 | Passed | Copy relies on repeated applications of block read and write, and so may suffer some consequences for larger files. Preliminary testing with various file sizes indicated a problem, and it seems to have been resolved, but it is something to keep an eye on.  Additionally, the use of isSameFile in the implementation presents a difference in behavior from the default. However, this concern was disregarded by the client, and so the implementation is considered valid. |
|  | Copy Test 2 | Passed | N/A |
| Get File Store | Get File Store Test 1 | Passed | N/A |
|  | Get File Store Test 2 | Passed | N/A |
| Is Same File | Is Same File Test 1 | Passed | N/A |
|  | Is Same File Test 2 | Passed | N/A |
|  | Is Same File Test 3 | Passed | N/A |
|  | Is Same File Test 4 | Passed | N/A |
| Delete | Delete Test 1 | Passed | N/A |
|  | Delete Test 2 | Passed | N/A |
|  | Delete Test 3 | Passed | N/A |
|  | Delete Test 4 | Passed | N/A |
|  | Delete Test 5 | Passed | N/A |
|  | Delete Test 6 | Passed | Testing indicated that delete may present a problem deleting hard links depending on the gluster volume’s caching settings. Specifically, while the file may be deleted, the volume’s caching settings may prevent a user from recreating the hard link after deletion.  Our client regards this as noteworthy, but not a flaw in the implementation. |

# 8: Glossary

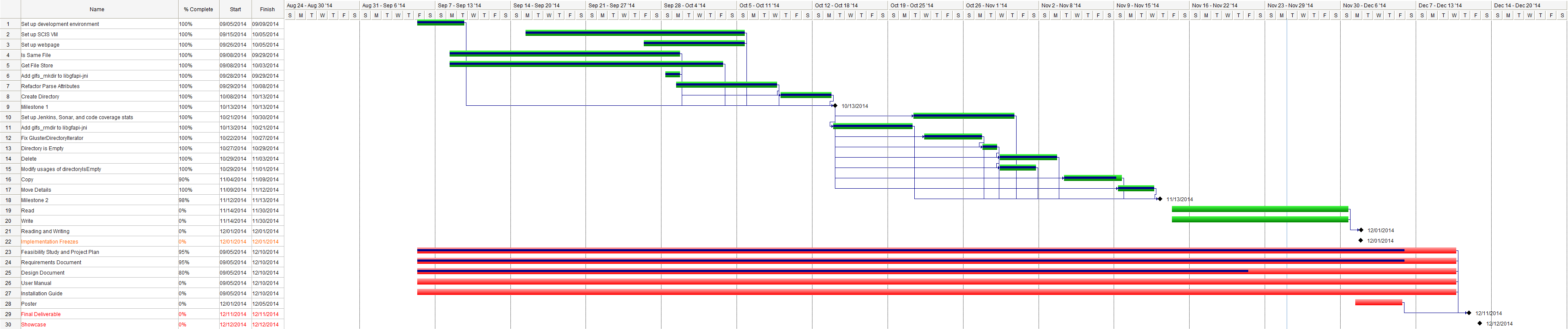
* **GlusterFS**: A scale-out, networked-attached storage platform
* **Java Native Interface (JNI)**: A programming framework which allows Java code to call and be called by native applications and libraries written in other languages
* **libgfapi**: A library for accessing data in GlusterFS
* **libgfapi-jni**: Java Native Interface bindings for the libgfapi
* **NIO.2**: Java 7’s enhanced file package which provides file system APIs
* **NIO**: Non-blocking I/O; also called asynchronous I/O

# 9: Appendix

This chapter consists of the following appendices:

* Appendix A – Project Schedule
* Appendix B – Use Cases
* Appendix C – Analysis Models
* Appendix D – Design Models
* Appendix E – Documented Class Interfaces and Constraints
* Appendix F – Documented Code for Test Drivers and Stubs
* Appendix G –Diary of Meetings

## 9.1: Appendix A – Project Schedule



Legend:

Orange: Important date

Green: Task

Diamond: Milestone

Red: Deliverable

## 9.2: Appendix B – Use Cases

The use cases described herein represent the functionality involved up to the second milestone of the project. Use cases will be elaborated on as the milestones are elaborated on throughout the project and as time allows for expansion of functionality.

|  |  |
| --- | --- |
| **Use Case ID** | **Copy (java-glusterfs-02)** |
| **Description** | Allows a user to copy a file or directory. |
| **Actor** | All Users |
| **Preconditions** | * User has a path to a file or directory to be copied. * User has a set of “copy options” that they desire. * User has a path to the target directory. * User has connected to a GlusterFS volume in their program. |
| **Details** | 1. The use case begins when the user calls the copy method in their program. 2. The system shall check if the desired copy options are supported. 3. The system shall check if the destination path exists. 4. The system shall check if the destination path is an empty directory. 5. The system shall copy the source file to the destination path based on the desired copy options. |
| **Postconditions** | The file or directory has been copied to the specified target location. |
| **Exceptions** | * The file already exists in the target location. * The directory already exists in the target location and it is not empty. * An I/O exception occurs. |

|  |  |
| --- | --- |
| **Use Case ID** | **Create Directory (java-glusterfs-03)** |
| **Description** | Allows a user to create a new directory. |
| **Actor** | All Users |
| **Preconditions** | * User has a path to the desired directory location. * User has connected to a GlusterFS volume in their program. |
| **Details** | 1. The use case begins when the user calls the createDirectory method in their program. 2. The system shall check if the path exists. 3. The system shall check if the parent of the path exists. 4. The system shall create a new directory at the location specified by the supplied path with a default set of permissions. |
| **Postconditions** | The directory has been created at the location specified by the path supplied by the user. |
| **Exceptions** | * A file with the same name requested by the user for the directory already exists. * The parent directory does not exist. * An I/O exception occurs. |

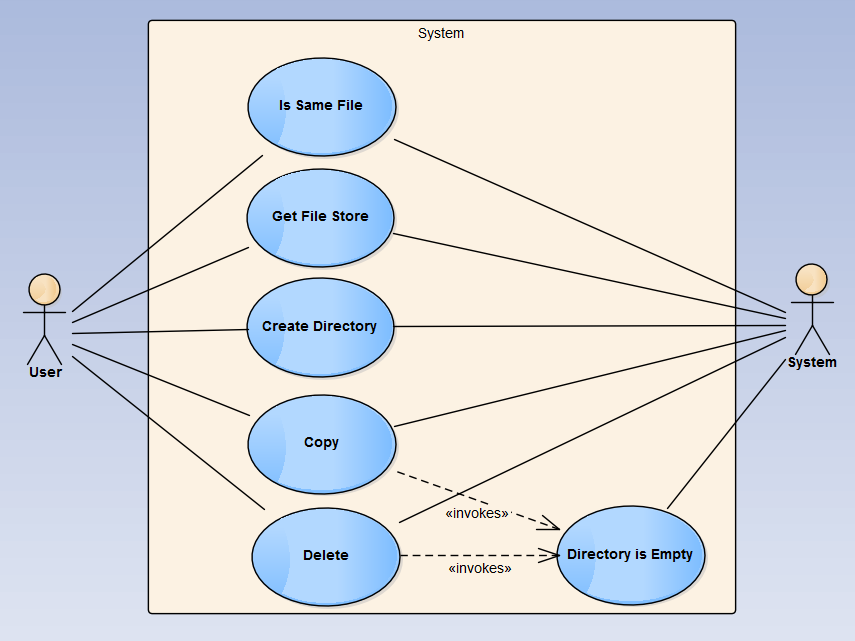
|  |  |
| --- | --- |
| **Use Case ID** | **Is Same File (java-glusterfs-04)** |
| **Description** | Allows a user to check if two files are the same. |
| **Actor** | All Users |
| **Preconditions** | * User has the paths to two files. * User has connected to a GlusterFS volume in their program. |
| **Details** | 1. The use case begins when the user calls the isSameFile method from within their program. 2. The system shall check if the two paths are the same. 3. The system shall check if the file systems of the two paths are the different. 4. The system shall check if both paths exist. 5. The system shall retrieve the inode number of both files. 6. The system shall check whether the inode numbers of both files are the same. 7. The system shall return whether or not the two inode numbers are the same. |
| **Postconditions** | The user knows if the two files are the same. |
| **Exceptions** | * An I/O exception occurs. * Either file does not exist. |

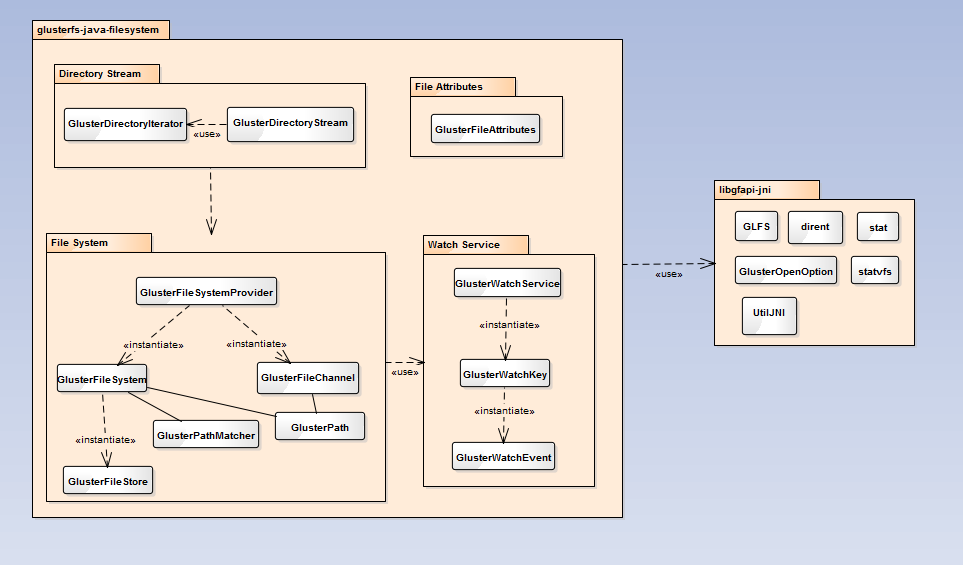
|  |  |
| --- | --- |
| **Use Case ID** | **Directory Is Empty (java-glusterfs-05)** |
| **Description** | Allows an actor to check if a directory is empty. |
| **Actor** | System |
| **Preconditions** | * Actor has a path to a directory. * Actor has access to a GlusterFS volume. |
| **Details** | 1. The use case begins when the actor calls the directoryIsEmpty method. 2. The system shall check if the path is a directory. 3. The system shall check if the directory has any entries. 4. The system shall return whether or not the directory has any entries. |
| **Postconditions** | The actor knows if the directory is empty. |
| **Exceptions** | * An I/O exception occurs. * The specified path does not exist. * The specified path is not a directory. |

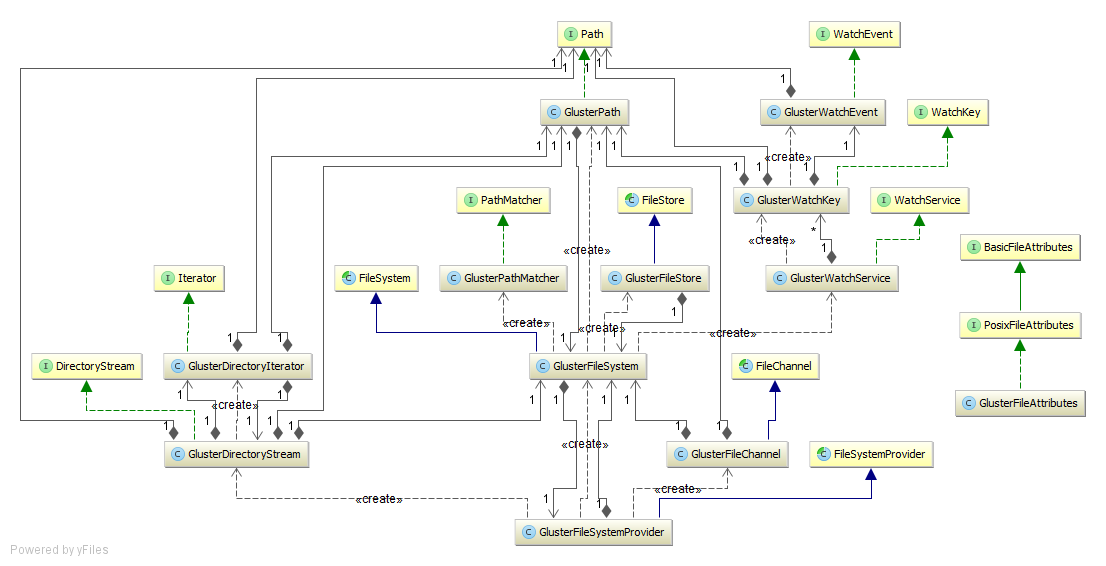
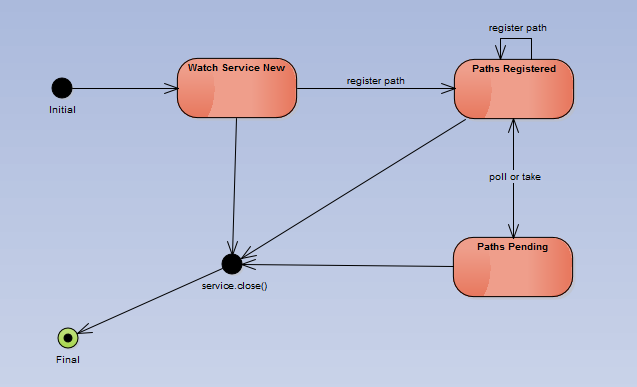
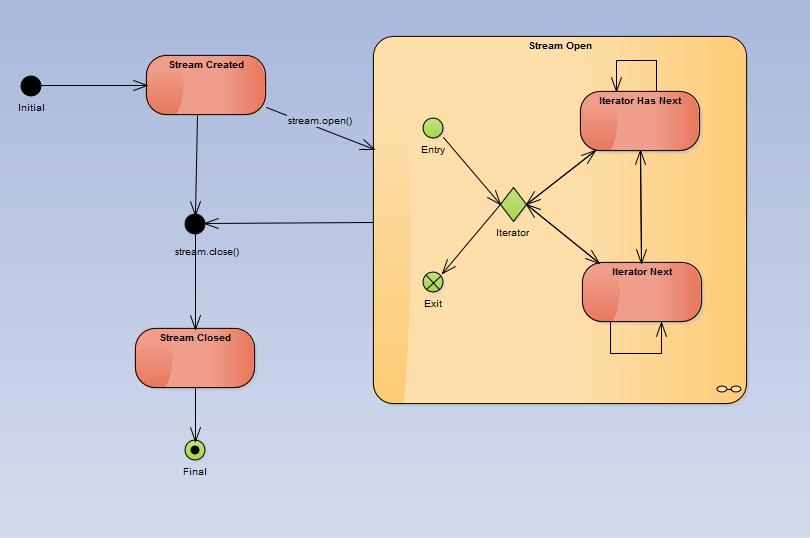
|  |  |
| --- | --- |
| **Use Case ID** | **Delete (java-glusterfs-01)** |
| **Description** | Allows a user to delete a file or directory. |
| **Actor** | All Users |
| **Preconditions** | * User has a path to a file or directory to be deleted. * User has connected to a GlusterFS volume in their program. |
| **Details** | 1. The use case begins when the user calls the delete method in their program. 2. The system shall check if the file is a directory. 3. The system shall check if the directory is empty. 4. The system shall delete the file or directory from the volume specified by the path provided. |
| **Postconditions** | The file has been deleted. |
| **Exceptions** | * The file does not exist. * The path specifies a directory and it is not empty. * An I/O exception occurs. |

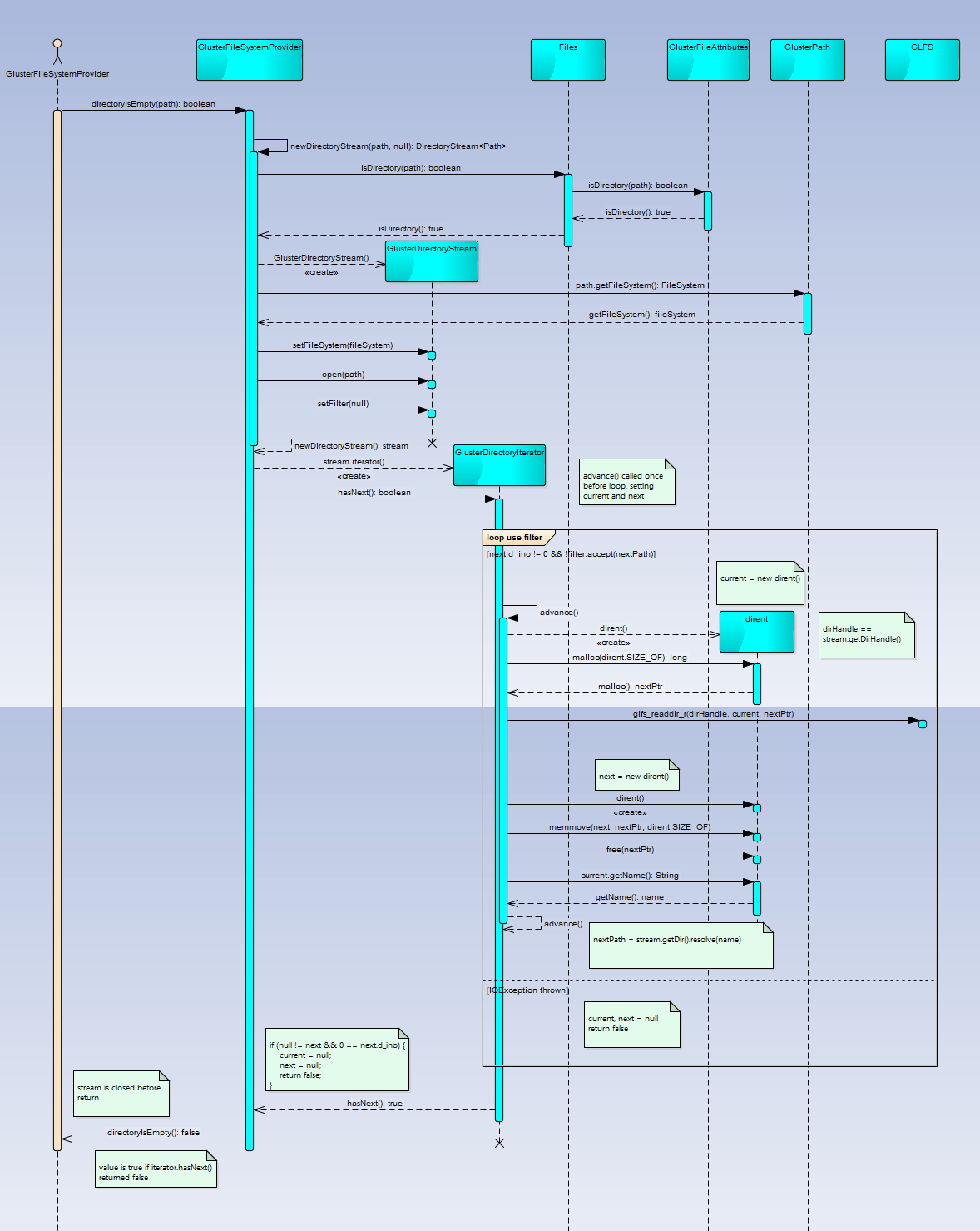
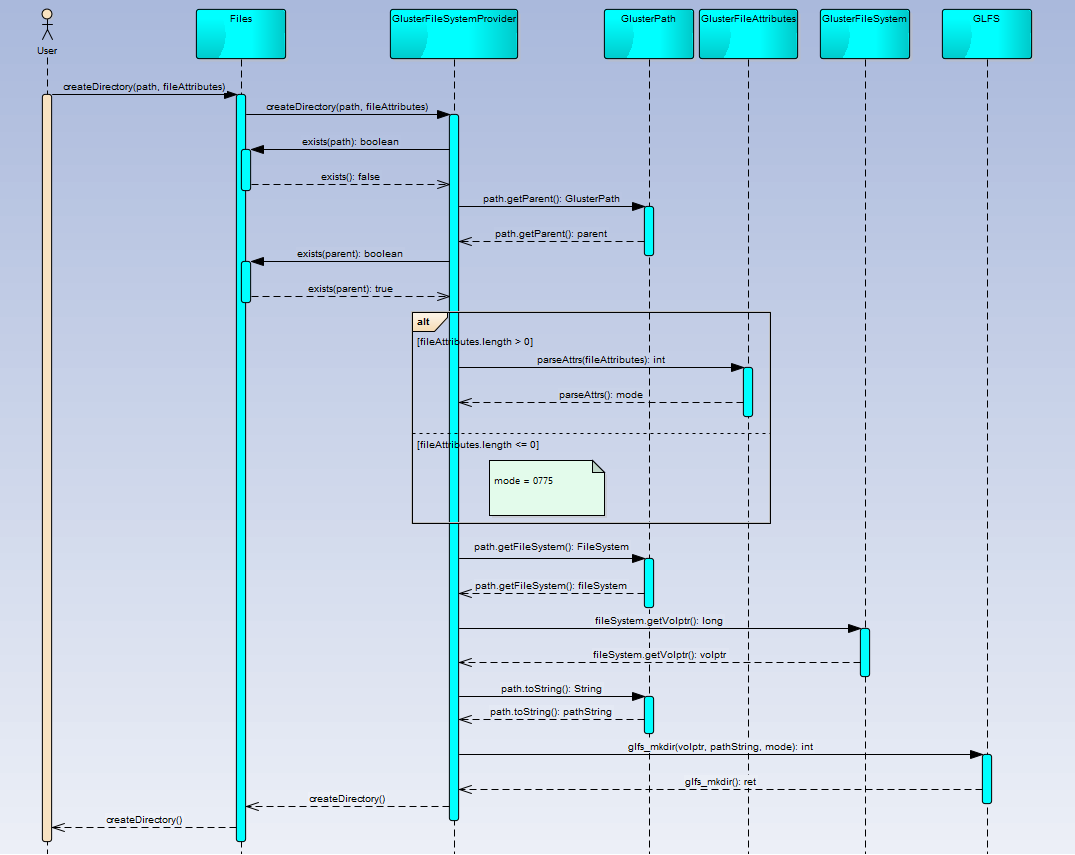
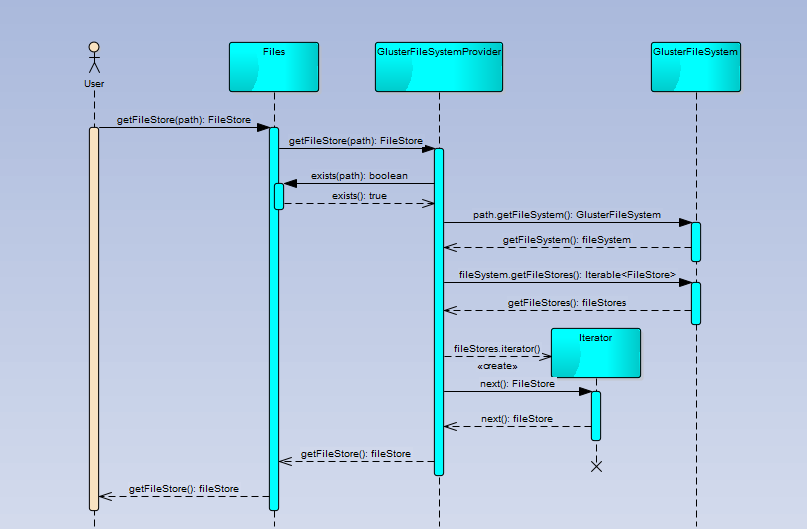
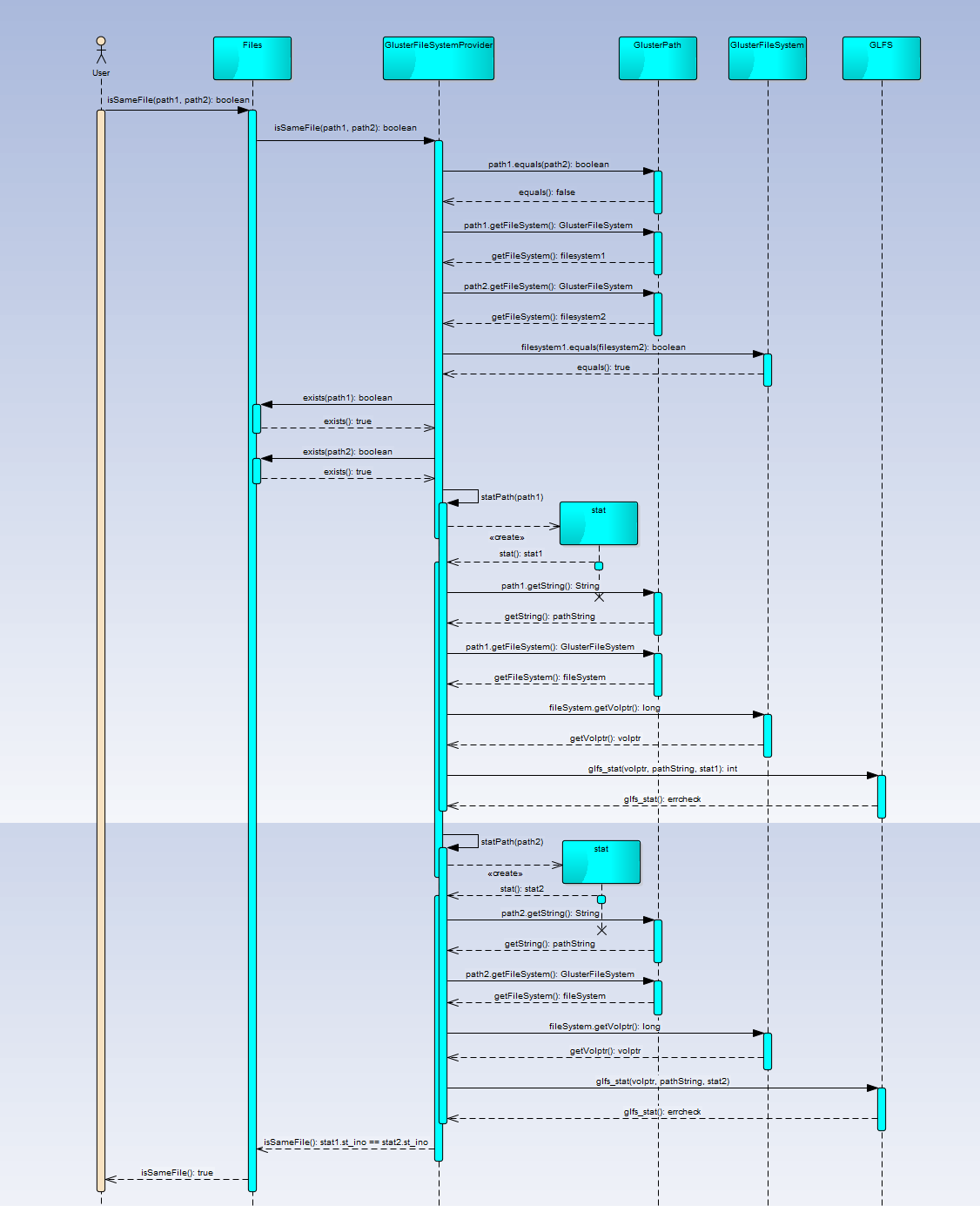
|  |  |
| --- | --- |
| **Use Case ID** | **Get File Store (java-glusterfs-06)** |
| **Description** | Allows a user to retrieve the file store of a file or directory. |
| **Actor** | All Users |
| **Preconditions** | * User has a path to a file or directory. * User has connected to a GlusterFS volume in their program. |
| **Details** | 1. The use case begins when the user calls the getFileStore method. 2. The system shall check if the file exists. 3. The system shall get the file system associated with the file. 4. The system shall get all of the file stores associated with the file system. 5. The system shall return a file store in that set. |
| **Postconditions** | The user has the file store associated with the file. |
| **Exceptions** | * The file does not exist. * An I/O exception occurs. |

## 9.3: Appendix C – Analysis Models





9.4 Appendix D – Design Models



## 9.5: Appendix D – Documented Class Interfaces and Constraints

public class GlusterFileAttributes implements PosixFileAttributes {

private static Map<Integer, PosixFilePermission> modeToPerms;

private static Map<PosixFilePermission, Integer> permsToMode;

private final int mode, uid, gid;

private final long size, atime, ctime, mtime, inode;

public static GlusterFileAttributes fromStat(stat stat);

public static int parseAttrs(FileAttribute<?>... attrs);

private static Map<PosixFilePermission, Integer> invertModeMap(Map<Integer, PosixFilePermission> modeToPerms);

@Override

public UserPrincipal owner();

@Override

public GroupPrincipal group();

@Override

public Set<PosixFilePermission> permissions();

@Override

public FileTime lastModifiedTime();

@Override

public FileTime lastAccessTime();

@Override

public FileTime creationTime();

@Override

public boolean isRegularFile();

@Override

public boolean isDirectory();

@Override

public boolean isSymbolicLink();

@Override

public boolean isOther();

@Override

public long size();

@Override

public Object fileKey();

}

public class GlusterDirectoryStream implements DirectoryStream<Path> {

private GlusterFileSystem fileSystem;

private long dirHandle;

private GlusterDirectoryIterator iterator;

private boolean closed;

private GlusterPath dir;

private DirectoryStream.Filter<? super Path> filter;

@Override

public Iterator<Path> iterator();

@Override

public void close() throws IOException;

public void open(GlusterPath path);

}

public class GlusterFileSystemProvider extends FileSystemProvider {

public static final String GLUSTER;

public static final int GLUSTERD\_PORT;

public static final String TCP;

private static Map<String, GlusterFileSystem> cache = new HashMap<String, GlusterFileSystem>();

@Override

public String getScheme();

@Override

public FileSystem newFileSystem(URI uri, Map<String, ?> stringMap) throws IOException;

String[] parseAuthority(String authority);

long glfsNew(String volname);

void glfsSetVolfileServer(String host, long volptr);

void glfsInit(String authorityString, long volptr);

@Override

public FileSystem getFileSystem(URI uri);

@Override

public Path getPath(URI uri);

@Override

public SeekableByteChannel newByteChannel(Path path, Set<? extends OpenOption> openOptions, FileAttribute<?>... fileAttributes) throws IOException;

@Override

public FileChannel newFileChannel(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs) throws IOException;

FileChannel newFileChannelHelper(Path path, Set<? extends OpenOption> options, FileAttribute<?>[] attrs) throws IOException;

@Override

public DirectoryStream<Path> newDirectoryStream(Path path, DirectoryStream.Filter<? super Path> filter) throws IOException;

@Override

public void createDirectory(Path path, FileAttribute<?>... fileAttributes) throws IOException;

@Override

public void delete(Path path) throws IOException;

@Override

public void copy(Path path, Path path2, CopyOption... copyOptions) throws IOException;

void copyFileAttributes(Path path, Path path2);

void copyFileContent(Path path, Path path2);

boolean directoryIsEmpty(Path path) throws IOException;

@Override

public void move(Path path, Path path2, CopyOption... copyOptions) throws IOException;

@Override

public boolean isSameFile(Path path, Path path2) throws IOException;

stat statPath(Path path) throws IOException;

@Override

public boolean isHidden(Path path) throws IOException;

@Override

public FileStore getFileStore(Path path) throws IOException;

@Override

public void checkAccess(Path path, AccessMode... accessModes) throws IOException;

private int modeInt(AccessMode m);

@Override

public <V extends FileAttributeView> V getFileAttributeView(Path path, Class<V> vClass, LinkOption... linkOptions);

@Override

public <A extends BasicFileAttributes> A readAttributes(Path path, Class<A> type, LinkOption... linkOptions) throws IOException;

@Override

public Map<String, Object> readAttributes(Path path, String s, LinkOption... linkOptions) throws IOException;

@Override

public void setAttribute(Path path, String s, Object o, LinkOption... linkOptions) throws IOException;

int close(long volptr);

long getTotalSpace(long volptr) throws IOException;

long getUsableSpace(long volptr) throws IOException;

long getUnallocatedSpace(long volptr) throws IOException;

}

public class GlusterWatchService implements WatchService {

public static final int MILLIS\_PER\_SECOND;

public static final int MILLIS\_PER\_MINUTE;

public static final int MILLIS\_PER\_HOUR;

public static final int MILLIS\_PER\_DAY;

public static long PERIOD;

private Set<GlusterWatchKey> paths;

private Set<GlusterWatchKey> pendingPaths;

private boolean running;

public WatchKey registerPath(GlusterPath path, WatchEvent.Kind... kinds);

@Override

public void close() throws IOException;

WatchKey popPending();

@Override

public WatchKey poll();

@Override

public WatchKey poll(long timeout, TimeUnit unit);

@Override

public WatchKey take();

long timeoutToMillis(long timeout, TimeUnit unit);

}

## 9.6: Appendix F – Documented Code for Test Drivers and Stubs

Integration testing of implemented methods:

public class Example {

public static FileSystemProvider getProvider(String scheme) {

for (FileSystemProvider fsp : FileSystemProvider.installedProviders()) {

if (fsp.getScheme().equals(scheme)) {

return fsp;

}

}

throw new IllegalArgumentException("No provider found for scheme: " + scheme);

}

public static void main(String[] args) throws URISyntaxException, IOException, InterruptedException {

Properties properties = new Properties();

properties.load(Example.class.getClassLoader().getResourceAsStream("example.properties"));

String glusterIP = properties.getProperty("glusterfs.server");

String volname = properties.getProperty("glusterfs.volume");

String mountStr = "gluster://" + glusterIP + ":" + volname + "/";

String dirStr = mountStr + "baz/";

String barStr = mountStr + "bar";

String fooStr = mountStr + "foo";

String foobarStr = mountStr + "foobar/";

String hardlinkStr = mountStr + "bar.hardlink";

String symlinkStr = mountStr + "bar.symlink";

String fakePathStr = mountStr + "fakePath";

Path dirPath = Paths.get(new URI(dirStr));

Path barPath = Paths.get(new URI(barStr));

Path fooPath = Paths.get(new URI(fooStr));

Path foobarPath = Paths.get(new URI(foobarStr));

Path hardlinkPath = Paths.get(new URI(hardlinkStr));

Path symlinkPath = Paths.get(new URI(symlinkStr));

try{

Files.createDirectory(dirPath);

System.out.println("Done; that was fast.");

} catch (IOException e) {

System.out.println("Could not create directory. Something went wrong.");

System.exit(-1);

}

try {

Files.createDirectory(dirPath);

} catch (FileAlreadyExistsException e) {

System.out.println("Failed to create directory: already exists. Good!");

}

Path fakePath = Paths.get(new URI(fakePathStr));

FileSystemProvider gfsp = barPath.getFileSystem().provider();

System.out.println("File store for bar in gluster provider: " + gfsp.getFileStore(barPath));

try {

System.out.println("File store for fakePath (this should not print): " + gfsp.getFileStore(fakePath));

} catch (IOException e) {

System.out.println("fakePath does not exist: good!");

}

System.out.println("Test of isSameFile:");

System.out.println("Is bar the same as bar? " + Files.isSameFile(barPath, barPath));

System.out.println("Is bar the same as foo? " + Files.isSameFile(barPath, fooPath));

System.out.println("Is bar the same as bar.hardlink? " + Files.isSameFile(barPath, hardlinkPath));

System.out.println("Is bar the same as bar.symlink? " + Files.isSameFile(barPath, symlinkPath));

try {

Files.delete(dirPath);

System.out.println("No problem.");

} catch (IOException e) {

System.out.println("Something went wrong! We're not supposed to be here!");

}

try {

Files.delete(dirPath);

System.out.println("If you are reading this, then I am a failure.");

} catch (IOException e) {

System.out.println("All's well.");

}

try {

Files.delete(fooPath);

System.out.println("Yep.");

} catch (IOException e) {

System.out.println("Nope.");

}

try {

Files.delete(foobarPath);

System.out.println("Yes. Bad!");

} catch (IOException e) {

System.out.println("No. Good!");

}

try {

Files.delete(symlinkPath);

System.out.println("Yep!");

} catch (IOException e) {

System.out.println("Nope!");

}

}

}

Unit tests written by us for subsystems we worked on:

package com.peircean.glusterfs;

import com.peircean.libgfapi\_jni.internal.GLFS;

import com.peircean.libgfapi\_jni.internal.structs.dirent;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.mockito.Spy;

import org.powermock.api.mockito.PowerMockito;

import org.powermock.core.classloader.annotations.PrepareForTest;

import org.powermock.core.classloader.annotations.SuppressStaticInitializationFor;

import org.powermock.modules.junit4.PowerMockRunner;

import java.nio.file.DirectoryStream;

import java.nio.file.Path;

import java.util.NoSuchElementException;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

@RunWith(PowerMockRunner.class)

@PrepareForTest({GLFS.class, GlusterDirectoryIterator.class, dirent.class})

@SuppressStaticInitializationFor("com.peircean.libgfapi\_jni.internal.structs.dirent")

public class GlusterDirectoryIteratorTest {

@Mock

private GlusterPath mockPath;

@Mock

private GlusterPath fakeResultPath;

@Mock

private GlusterFileSystem mockFileSystem;

@Mock

private GlusterDirectoryStream mockStream;

@Mock

private dirent mockCurrentDirent;

@Mock

private dirent mockNextDirent;

@Mock

private DirectoryStream.Filter<? super Path> mockFilter;

@Spy

private GlusterDirectoryIterator iterator = new GlusterDirectoryIterator();

private long dirHandle = 12345l;

@Test

public void testAdvance\_whenNormalEntry() throws Exception {

advanceHelper(0, true);

}

@Test

public void testAdvance\_skipSpecialEntryCurrent() throws Exception {

advanceHelper(1, true);

}

@Test

public void testAdvance\_skipSpecialEntryParent() throws Exception {

advanceHelper(1, false);

}

@Test

public void testAdvance\_skipSpecialEntryCurrentAndParent() throws Exception {

advanceHelper(2, true);

}

/\*

\* Type 0 = advancing on normal entry, so the boolean current value does not matter

\* Type 1 = advancing on either special entry "." or "..", differentiated by boolean current

\* Type 2 = advancing on both special entries "." and "..", so boolean current value does not matter

\*/

private void advanceHelper(int type, boolean current) throws Exception {

doReturn(dirHandle).when(mockStream).getDirHandle();

iterator.setStream(mockStream);

switch (type) {

case 1:

PowerMockito.whenNew(dirent.class).withNoArguments().thenReturn(mockCurrentDirent, mockNextDirent,

mockCurrentDirent, mockNextDirent);

break;

case 2:

PowerMockito.whenNew(dirent.class).withNoArguments().thenReturn(mockCurrentDirent, mockNextDirent,

mockCurrentDirent, mockNextDirent,

mockCurrentDirent, mockNextDirent);

break;

default:

PowerMockito.whenNew(dirent.class).withNoArguments().thenReturn(mockCurrentDirent, mockNextDirent);

break;

}

long nextPtr = 4444l;

PowerMockito.mockStatic(dirent.class);

Mockito.when(dirent.malloc(dirent.SIZE\_OF)).thenReturn(nextPtr);

PowerMockito.doNothing().when(dirent.class);

dirent.memmove(mockNextDirent, nextPtr, dirent.SIZE\_OF);

PowerMockito.doNothing().when(dirent.class);

dirent.free(nextPtr);

PowerMockito.mockStatic(GLFS.class);

PowerMockito.when(GLFS.glfs\_readdir\_r(dirHandle, mockCurrentDirent, nextPtr)).thenReturn(0);

Mockito.doReturn(mockPath).when(mockStream).getDir();

String stringPath = "foo";

switch (type) {

case 1:

if (current) {

when(mockCurrentDirent.getName()).thenReturn(".", stringPath);

} else {

when(mockCurrentDirent.getName()).thenReturn("..", stringPath);

}

break;

case 2:

when(mockCurrentDirent.getName()).thenReturn(".", "..", stringPath);

break;

default:

when(mockCurrentDirent.getName()).thenReturn(stringPath);

break;

}

when(mockPath.resolve(any(String.class))).thenReturn(fakeResultPath);

iterator.advance();

assertEquals(fakeResultPath, iterator.getNextPath());

verify(mockPath).resolve(stringPath);

switch (type) {

case 1:

verify(mockCurrentDirent, times(2)).getName();

if (current) {

verify(mockPath).resolve(".");

} else {

verify(mockPath).resolve("..");

}

verify(mockStream, times(2)).getDir();

verify(mockStream, times(2)).getDirHandle();

PowerMockito.verifyNew(dirent.class, times(4)).withNoArguments();

PowerMockito.verifyStatic(times(2));

dirent.memmove(mockNextDirent, nextPtr, dirent.SIZE\_OF);

PowerMockito.verifyStatic(times(2));

dirent.free(nextPtr);

break;

case 2:

verify(mockCurrentDirent, times(3)).getName();

verify(mockPath).resolve(".");

verify(mockPath).resolve("..");

verify(mockStream, times(3)).getDir();

verify(mockStream, times(3)).getDirHandle();

PowerMockito.verifyNew(dirent.class, times(6)).withNoArguments();

PowerMockito.verifyStatic(times(3));

dirent.memmove(mockNextDirent, nextPtr, dirent.SIZE\_OF);

PowerMockito.verifyStatic(times(3));

dirent.free(nextPtr);

break;

default:

verify(mockCurrentDirent).getName();

verify(mockStream).getDir();

verify(mockStream).getDirHandle();

PowerMockito.verifyNew(dirent.class, times(2)).withNoArguments();

PowerMockito.verifyStatic();

dirent.memmove(mockNextDirent, nextPtr, dirent.SIZE\_OF);

PowerMockito.verifyStatic();

dirent.free(nextPtr);

break;

}

}

}

package com.peircean.glusterfs;

import com.peircean.libgfapi\_jni.internal.GLFS;

import com.peircean.libgfapi\_jni.internal.structs.stat;

import com.peircean.libgfapi\_jni.internal.structs.statvfs;

import junit.framework.TestCase;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.mockito.Spy;

import org.powermock.api.mockito.PowerMockito;

import org.powermock.core.classloader.annotations.PrepareForTest;

import org.powermock.modules.junit4.PowerMockRunner;

import java.io.IOException;

import java.net.URI;

import java.net.URISyntaxException;

import java.nio.channels.ByteChannel;

import java.nio.channels.FileChannel;

import java.nio.file.\*;

import java.nio.file.attribute.DosFileAttributes;

import java.nio.file.attribute.FileAttribute;

import java.util.\*;

import static org.mockito.Mockito.doNothing;

import static org.mockito.Mockito.doReturn;

import static org.mockito.Mockito.doThrow;

import static org.mockito.Mockito.\*;

import static org.mockito.Mockito.when;

import static org.powermock.api.mockito.PowerMockito.mockStatic;

import static org.powermock.api.mockito.PowerMockito.verifyStatic;

import static org.powermock.api.mockito.PowerMockito.whenNew;

import static org.powermock.api.mockito.PowerMockito.verifyNew;

@RunWith(PowerMockRunner.class)

@PrepareForTest({GLFS.class, Files.class, GlusterFileSystemProvider.class, GlusterFileChannel.class, GlusterFileAttributes.class,

GlusterDirectoryStream.class, GlusterFileSystem.class})

public class GlusterFileSystemProviderTest extends TestCase {

public static final String SERVER = "hostname";

public static final String VOLNAME = "testvol";

@Mock

private GlusterFileSystem mockFileSystem;

@Mock

private GlusterFileSystem differentMockFileSystem;

@Mock

private GlusterPath mockPath;

@Mock

private GlusterPath targetPath;

@Mock

private GlusterFileChannel mockChannel;

@Mock

private GlusterDirectoryIterator mockIterator;

@Mock

private GlusterDirectoryStream mockStream;

@Mock

private DirectoryStream.Filter<? super Path> mockFilter;

@Spy

private GlusterFileSystemProvider provider = new GlusterFileSystemProvider();

@Test(expected = NoSuchFileException.class)

public void testDelete\_whenFileDoesNotExist() throws IOException {

mockStatic(Files.class);

when(Files.exists(mockPath)).thenReturn(false);

provider.delete(mockPath);

}

@Test(expected = DirectoryNotEmptyException.class)

public void testDelete\_whenDirectoryIsNotEmpty() throws IOException {

mockStatic(Files.class);

when(Files.exists(mockPath)).thenReturn(true);

when(Files.isDirectory(mockPath)).thenReturn(true);

doReturn(false).when(provider).directoryIsEmpty(mockPath);

provider.delete(mockPath);

}

@Test(expected = IOException.class)

public void testDelete\_whenDirectoryAndFailing() throws IOException {

delete\_helper(true, false);

}

@Test(expected = IOException.class)

public void testDelete\_whenFileAndFailing() throws IOException {

delete\_helper(false, false);

}

@Test

public void testDelete\_whenFileAndSuccess() throws IOException {

delete\_helper(false, true);

}

@Test

public void testDelete\_whenDirectoryAndSuccess() throws IOException {

delete\_helper(true, true);

}

private void delete\_helper(boolean directory, boolean success) throws IOException {

long volptr = 1234L;

String path = "/foo";

mockStatic(Files.class);

mockStatic(GLFS.class);

when(Files.exists(mockPath)).thenReturn(true);

doReturn(mockFileSystem).when(mockPath).getFileSystem();

doReturn(volptr).when(mockFileSystem).getVolptr();

doReturn(path).when(mockPath).toString();

if (directory) {

when(Files.isDirectory(mockPath)).thenReturn(true);

doReturn(true).when(provider).directoryIsEmpty(mockPath);

if (success) {

when(GLFS.glfs\_rmdir(volptr, path)).thenReturn(0);

} else {

when(GLFS.glfs\_rmdir(volptr, path)).thenReturn(-1);

}

} else {

when(Files.isDirectory(mockPath)).thenReturn(false);

if (success) {

when(GLFS.glfs\_unlink(volptr, path)).thenReturn(0);

} else {

when(GLFS.glfs\_unlink(volptr, path)).thenReturn(-1);

}

}

provider.delete(mockPath);

if (success) {

verifyStatic();

Files.exists(mockPath);

verifyStatic();

Files.isDirectory(mockPath);

verify(mockPath).getFileSystem();

verify(mockFileSystem).getVolptr();

if (directory) {

verify(provider).directoryIsEmpty(mockPath);

verifyStatic();

GLFS.glfs\_rmdir(volptr, path);

} else {

verifyStatic();

GLFS.glfs\_unlink(volptr, path);

}

}

}

@Test(expected = FileAlreadyExistsException.class)

public void testCopyFile\_whenTargetExists\_andNoReplaceExisting() throws IOException {

Path targetPath = mockPath.resolveSibling("copy");

mockStatic(Files.class);

when(Files.exists(targetPath)).thenReturn(true);

provider.copy(mockPath, targetPath);

}

@Test(expected = DirectoryNotEmptyException.class)

public void testCopyFile\_whenTargetDirNotEmpty\_andReplaceExisting() throws IOException {

Path targetPath = mockPath.resolveSibling("copy");

mockStatic(Files.class);

when(Files.isDirectory(targetPath)).thenReturn(true);

doReturn(false).when(provider).directoryIsEmpty(targetPath);

provider.copy(mockPath, targetPath, StandardCopyOption.REPLACE\_EXISTING);

}

@Test

public void testCopyFile\_whenTargetDoesNotExist() throws IOException {

helperCopyFile(false);

}

@Test

public void testCopyFile\_whenTargetDoesNotExist\_andCopyAttributes() throws IOException {

helperCopyFile(true);

}

void helperCopyFile(boolean attributes) throws IOException {

Path targetPath = mockPath.resolveSibling("copy");

mockStatic(Files.class);

when(Files.isDirectory(targetPath)).thenReturn(false);

when(Files.exists(targetPath)).thenReturn(false);

when(Files.createFile(targetPath)).thenReturn(targetPath);

doNothing().when(provider).copyFileContent(mockPath, targetPath);

if (attributes) {

doNothing().when(provider).copyFileAttributes(mockPath, targetPath);

provider.copy(mockPath, targetPath, StandardCopyOption.COPY\_ATTRIBUTES);

} else {

provider.copy(mockPath, targetPath);

}

verify(provider).copyFileContent(mockPath, targetPath);

if (attributes) {

verify(provider).copyFileAttributes(mockPath, targetPath);

}

verifyStatic();

Files.isDirectory(targetPath);

Files.exists(targetPath);

Files.createFile(targetPath);

}

@Test(expected = FileAlreadyExistsException.class)

public void testCreateDirectory\_whenFileOrDirectoryExists() throws IOException {

mockStatic(Files.class);

when(Files.exists(mockPath)).thenReturn(true);

provider.createDirectory(mockPath);

}

@Test(expected = IOException.class)

public void testCreateDirectory\_whenParentDirectoryDoesNotExist() throws IOException {

mockStatic(Files.class);

GlusterPath parentPath = Mockito.mock(GlusterPath.class);

doReturn(parentPath).when(mockPath).getParent();

when(Files.exists(parentPath)).thenReturn(false);

provider.createDirectory(mockPath);

}

@Test(expected = IOException.class)

public void testCreateDirectory\_whenCannotCreateDirectory() throws IOException {

helperCreateDirectory(true);

}

@Test

public void testCreateDirectory() throws IOException {

helperCreateDirectory(false);

}

private void helperCreateDirectory(boolean errorHappens) throws IOException {

mockStatic(Files.class);

when(Files.exists(mockPath)).thenReturn(false);

GlusterPath parentPath = Mockito.mock(GlusterPath.class);

doReturn(parentPath).when(mockPath).getParent();

when(Files.exists(parentPath)).thenReturn(true);

int mode = 0775; //using default file attribute to avoid testing parseAttrs redundantly

long volptr = 1234L;

String pathString = "foo";

doReturn(pathString).when(mockPath).toString();

doReturn(mockFileSystem).when(mockPath).getFileSystem();

doReturn(volptr).when(mockFileSystem).getVolptr();

mockStatic(GLFS.class);

int ret = 0;

if (errorHappens) {

ret = -1;

}

when(GLFS.glfs\_mkdir(volptr, pathString, mode)).thenReturn(ret);

provider.createDirectory(mockPath);

if (!errorHappens) {

verifyStatic();

GLFS.glfs\_mkdir(volptr, pathString, mode);

verifyStatic();

Files.exists(mockPath);

verifyStatic();

Files.exists(parentPath);

verify(mockPath).getFileSystem();

verify(mockFileSystem).getVolptr();

verify(mockPath).getParent();

}

}

@Test

public void testIsSameFile\_whenSamePaths() throws IOException {

doReturn("/").when(mockFileSystem).getSeparator();

GlusterPath path = new GlusterPath(mockFileSystem, "/foo/bar");

boolean ret = provider.isSameFile(path, path);

assertTrue(ret);

verify(mockFileSystem, times(3)).getSeparator();

}

@Test

public void testIsSameFile\_whenPathsEqual() throws IOException {

doReturn("/").when(mockFileSystem).getSeparator();

GlusterPath path1 = new GlusterPath(mockFileSystem, "/foo/bar");

GlusterPath path2 = new GlusterPath(mockFileSystem, "/foo/bar");

boolean ret = provider.isSameFile(path1, path2);

assertTrue(ret);

verify(mockFileSystem, times(6)).getSeparator();

}

@Test

public void testIsSameFile\_whenFilesystemsDiffer() throws IOException {

GlusterPath path = Mockito.mock(GlusterPath.class);

doReturn(mockFileSystem).when(mockPath).getFileSystem();

doReturn(differentMockFileSystem).when(path).getFileSystem();

boolean ret = provider.isSameFile(mockPath, path);

assertFalse(ret);

verify(mockPath).getFileSystem();

verify(path).getFileSystem();

}

@Test

public void testIsSameFile\_whenDifferent() throws Exception {

//in the case of a copy of a file

//different file paths and different inode numbers (asserting false)

isSameFile\_helper(false);

}

@Test

public void testIsSameFile\_whenSame() throws Exception {

//in the case of hardlinks and symlinks

//different file paths and identical inode numbers (asserting true)

isSameFile\_helper(true);

}

private void isSameFile\_helper(boolean same) throws Exception {

GlusterPath glusterPath = Mockito.mock(GlusterPath.class);

GlusterFileSystem actualFileSystem = new GlusterFileSystem(provider, "foohost", "volfoo", 1234L);

doNothing().when(provider).guardFileExists(mockPath);

doNothing().when(provider).guardFileExists(glusterPath);

doReturn(actualFileSystem).when(mockPath).getFileSystem();

doReturn(actualFileSystem).when(glusterPath).getFileSystem();

long sameIno = 222L;

stat stat1 = new stat();

stat1.st\_ino = sameIno;

long differentIno = 444L;

stat stat2 = new stat();

if (same) {

stat2.st\_ino = sameIno;

} else {

stat2.st\_ino = differentIno;

}

doReturn(stat1).when(provider).statPath(glusterPath);

doReturn(stat2).when(provider).statPath(mockPath);

boolean ret = provider.isSameFile(glusterPath, mockPath);

assertTrue(same == ret);

verify(mockPath).getFileSystem();

verify(glusterPath).getFileSystem();

verify(provider).statPath(glusterPath);

verify(provider).statPath(mockPath);

verify(provider).guardFileExists(mockPath);

verify(provider).guardFileExists(glusterPath);

}

@Test(expected = IOException.class)

public void testStatPath\_whenItFails() throws IOException {

long volptr = 1234L;

stat stat = new stat();

mockStatic(GLFS.class);

String pathString = "/foo";

when(GLFS.glfs\_stat(volptr, pathString, stat)).thenReturn(-1);

doReturn(mockFileSystem).when(mockPath).getFileSystem();

doReturn(volptr).when(mockFileSystem).getVolptr();

doReturn(pathString).when(mockPath).getString();

provider.statPath(mockPath);

}

@Test

public void testStatPath() throws Exception {

long volptr = 1234L;

stat stat = new stat();

whenNew(stat.class).withNoArguments().thenReturn(stat);

mockStatic(GLFS.class);

String pathString = "/foo";

when(GLFS.glfs\_stat(volptr, pathString, stat)).thenReturn(0);

doReturn(mockFileSystem).when(mockPath).getFileSystem();

doReturn(volptr).when(mockFileSystem).getVolptr();

doReturn(pathString).when(mockPath).getString();

stat ret = provider.statPath(mockPath);

assertTrue(ret == stat);

verify(mockPath).getFileSystem();

verify(mockFileSystem).getVolptr();

verify(mockPath).getString();

verifyStatic();

GLFS.glfs\_stat(volptr, pathString, stat);

verifyNew(stat.class).withNoArguments();

}

@Test

public void testGetFileStore\_whenFileExists() throws IOException {

mockStatic(Files.class);

when(Files.exists(mockPath)).thenReturn(true);

doReturn(mockFileSystem).when(mockPath).getFileSystem();

GlusterFileStore fileStore = new GlusterFileStore(mockFileSystem);

List<FileStore> stores = new ArrayList<>();

stores.add(fileStore);

Iterable<FileStore> iterable = Collections.unmodifiableList(stores);

doReturn(iterable).when(mockFileSystem).getFileStores();

FileStore retFileStore = provider.getFileStore(mockPath);

assertEquals(retFileStore, fileStore);

verify(mockPath).getFileSystem();

verify(mockFileSystem).getFileStores();

verifyStatic();

Files.exists(mockPath);

}

@Test(expected = NoSuchFileException.class)

public void testGetFileStore\_whenFileDoesNotExist() throws IOException {

mockStatic(Files.class);

when(Files.exists(mockPath)).thenReturn(false);

provider.getFileStore(mockPath);

}

@Test

public void testDirectoryIsEmpty\_whenEmpty() throws Exception {

directoryIsEmpty\_helper(true);

}

@Test

public void testDirectoryIsEmpty\_whenNotEmpty() throws Exception {

directoryIsEmpty\_helper(false);

}

private void directoryIsEmpty\_helper(boolean empty) throws Exception {

doReturn(mockStream).when(provider).newDirectoryStream(mockPath, null);

doReturn(mockIterator).when(mockStream).iterator();

if (empty) {

doReturn(false).when(mockIterator).hasNext();

} else {

doReturn(true).when(mockIterator).hasNext();

}

boolean result = provider.directoryIsEmpty(mockPath);

if (empty) {

assertTrue(result);

} else {

assertFalse(result);

}

verify(provider).newDirectoryStream(mockPath, null);

verify(mockStream).iterator();

verify(mockIterator).hasNext();

}

}

## 9.7: Appendix G – Diary of Meetings

**Begin meeting**: 9/3/14 7:15PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Personal introductions
* Overview of GlusterFS
* Overview of development environment
* Current state of glusterfs-java-filesystem
* Demos detailing functionality and unit tests
* Discuss extent of project (resulting in timeline)

**End meeting**: 9:00PM

**Next meeting date**: 9/7/14 3PM

**Next meeting agenda**: Set up development environment, sort out development logistics (physical and management perspectives)

**Start meeting**: 9/7/14 1:55PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Forked GlusterFS-Java-Filesystem repository into FIU SCIS organization
* Overview of Maven commands
* Set up development environment
  + Made shell script to run IntelliJ with correct Java 7 JDK
  + Built libgfapi-jni with Maven
  + Built glusterfs-java-filesystem with Maven
  + Ran glusterfs-java-filesystem-example with Maven
  + Ran/Debugged glusterfs-java-filesystem tests with IDEA
  + Ran/Debugged glusterfs-java-filesystem-example with IDEA
* Broke up milestones into tasks
* Discussed class diagrams and use cases

**End meeting**: 5:10PM

**Next meeting date**: 9/14/14 3PM

**Next meeting agenda**: Resolve issues with boot testing VM with Vagrant, discuss progress and position in timeline, and identify further tasks.

**Start meeting**: 9/14/14 3:40PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed shortcuts to make the development process more efficient in IntelliJ IDEA
* Concluded that toFile() should not be implemented in the project
* Discussed how relativize and resolve work in the library
* Overview of Mockito, Lombok plugin, and PowerMock syntax for JUnit testing
* Discussed the acceptance process to be used in the project

**End meeting**: 6:10PM

**Next meeting date**: 9/21/14 4PM

**Next meeting agenda**: Identify further tasks, discuss any issues with JUnit testing, and discuss the acceptance of any implemented and tested features.

**Start meeting**: 9/17/14 6:45PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Addressed questions concerning the creation of JUnit tests
* Overview of GlusterFS and how it works
* Decided on the structure of the git repository
* Discussed whether the SCIS virtual machine would suit our needs
* Overview of Jenkins and continuous integration

**End meeting**: 8:30PM

**Next meeting date**: 9/21/14 4PM

**Next meeting agenda**: Discuss the acceptance of the isSameFile and getFileStore features along with any questions concerning JUnit testing.

**Start meeting**: 9/21/14 3:45PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed the acceptance of the two completed features (issamefile and getfilestore)
* Overview of pull requests
* Discussed changes needed to be implemented in libgfapi-jni project for the upcoming createDirectory and directoryIsEmpty features
* Discussed integration testing in the libgfapi-jni project
* Overview of file attributes
* Discussed preferred programming practices for this project
* Overview of HawtJNI

**End meeting**: 5:30PM

**Next meeting date**: 9/28/14 4PM

**Next meeting agenda**: Discuss the next round of features set for acceptance along with any questions concerning testing in the libgfapi-jni project.

**Start meeting**: 9/28/14 7:00PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed how to parse file attributes and how to refactor our initial implementation to best accomplish this
* Solved the issue of glfs\_mkdir not being recognized in GlusterFileSystemProvider
* Discussed a procedure to follow for the next time that we encounter an issue with the snapshot and missing dependencies
* Discussed how to best demo our work and agreed to use Jenkins and Sonar
* Discussed how to fix issues encountered in getting our development environment to run in FIU's SCIS virtual machine

**End meeting**: 8:15PM

**Next meeting date**: 10/5/14 4PM

**Next meeting agenda**: Discuss the next round of features set for acceptance, and determine how to tie Jenkins and Sonar to our project.

**Start meeting**: 10/13/14 6:50PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed how to use and set up Jenkins with the Java GlusterFS project
* Discussed issue where createDirectory was resulting in an error on some machines
* Discussed how Java handles the default permissions for a new file when no file attributes are provided
* Discussed the acceptance of the createDirectory feature

**End meeting**: 8:50PM

**Next meeting date**: 10/20/14 7PM

**Next meeting agenda**: Discuss the next round of features set for acceptance as well as the current status of the project documents.

**Start meeting**: 10/20/14 7:10PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed a potential issue with the current GlusterDirectoryIterator implementation where the special entries "." and ".." were being returned (contrary to the behavior of the default provider)
* Determined the best fix for the GlusterDirectoryIterator issue
* Discussed the current implementation of isDirectoryEmpty
* Discussed a potential issue with calls to verifyStatic() in the JUnit tests
* Discussed how to set up Jenkins on the FIU VM

**End meeting**: 8:30PM

**Next meeting date**: 10/26/14 7PM

**Next meeting agenda**: Discuss the next round of features set for acceptance as well as the current status of the project documents. Additionally, possible modifications to the Jenkins and Sonar profiles will be addressed.

**Start meeting**: 10/29/14 7:10PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed how to set up the Atlassian Clover code coverage tool with Jenkins and Sonar
* Discussed possible tools and locations for filming our videos
* Discussed the current state of the documentation, including class diagrams and sequence diagrams
* Discussed the extent to which security and file permissions will be implemented and enforced

**End meeting**: 8:10PM

**Next meeting date**: 11/3/14 7PM

**Next meeting agenda**: Discuss the next round of features set for acceptance as well as the current status of the project documents and videos. Additionally, brainstorm real-world scenarios where our Java library can be used.

**Start meeting**: 11/12/14 6:45PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed the pull request for dirIsEmpty branch and any necessary changes to the current implementation or tests. Decided to modify if statements in advanceHelper method to switch cases.
* Discussed the move feature (specifically, how its implementation differs from spec) and decided that it is only missing a throws clause to be complete.
* Discussed the copy feature and how file attributes would be copied over (using chmod, chown).
* Discussed utilizing the platform-specific umask value in the logic of create directory. Decided that it would be best to not do this.
* Changed the priority of (1) user and group name and (2) reading and writing features. Reading/writing has a higher priority now, and user and group name feature will be done only if time permits.
* Discussed what aspects of the reading and writing features will need to be prioritized.
* Discussed how the user and group names feature would be implemented using a Linux UserPrincipleLookupService.

**End meeting**: 8:25PM

**Next meeting date**: 11/17/14 7PM

**Next meeting agenda**: Discuss the implementations and tests of delete, copy and move, as well as discuss the current status of the project documents and videos.

**Start meeting**: 6:45PM

**Location**: Picture Marketing office in Doral

**In attendance**: Louis Zuckerman, Maylem Gonzalez, Ian Herbig

**Summary of events**:

* Discussed the architectural and design patterns found in the Java-GlusterFS project. Determined that the following six patterns apply: Service Provider Interface, Abstract Factory, Object Pool, Singleton, Iterator, Fluent Interface, and Prototype.
* Determined that time should currently not be spent on trying to get Cobertura to work with our project.
* Discussed the use of isSameFile() in the move feature and how it results in a difference from specs. Decided that a note should be added to detail this change in behavior for future consideration.
* Discussed the use of ByteBuffer in reading/writing bytes in the project.
* Discussed the implementation of write(ByteBuffer src) and determined that the copy feature needs to be put on hold until the write implementation is modified to spec and bugs with the current implementation are fixed.
* Decided that UtilJNI should be made public instead of package protected.

**End meeting**: 8:15PM

10: References

|  |  |
| --- | --- |
| [1] | L. Zuckerman, "GitHub Repository for Java GlusterFS," [Online]. Available: https://github.com/semiosis/glusterfs-java-filesystem. [Accessed 5 9 2014]. |
| [2] | J. Darcy, "GlusterFS Internals and Directions," in *Red Hat Summit*, Boston, 2013. |
| [3] | "File I/O (Featuring NIO.2)," Oracle, [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/io/fileio.html. [Accessed 5 9 2014]. |
| [4] | "Java SE 7 Java Native Interface-related APIs and Developer Guides," Oracle, [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/. [Accessed 5 9 2014]. |