Tessella Project Number: 6870

**Tessella plc** 26 The Quadrant, Abingdon Science Park, Abingdon, Oxfordshire, OX14 3YS, UK

T: +44 (0)1235 555511 | F: +44 (0)1235 553301 | E: info@tessella.com | www.tessella.com

Client: RAL & ORNL

Mantid

File Finding Improvements Design Document



Document Control

This document is under document control. All members of the distribution list will receive updated copies whenever alterations are made. All other copies are uncontrolled.

Document Owner

|  |  |  |
| --- | --- | --- |
| Name | Role | Company |
| Nicholas Draper | Owner | Tessella |

Document Approval

|  |  |  |
| --- | --- | --- |
| Name | Approval Role | Company |
|  |  |  |

Distribution

|  |  |  |
| --- | --- | --- |
| Name | Role | Company |
| Mantid GIT repository | Filing | Mantid Project |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Author | Issue | Date | Description |
| Nicholas Draper | V1.R1.M0 | 23-Apr-2013 | 1st version written |

References

|  |  |  |  |
| --- | --- | --- | --- |
| Document | Ref | Date | Details & Issue |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 4](#_Toc357176636)

[1.1 Purpose of this Document 4](#_Toc357176637)

[1.2 Scope of this Document 4](#_Toc357176638)

[1.3 Context of this Issue 4](#_Toc357176639)

[1.4 Definition of Terms 4](#_Toc357176640)

[2 Current File Searching Overview 5](#_Toc357176641)

[2.1 Python interface 5](#_Toc357176642)

[2.2 Requirements for file searching 6](#_Toc357176643)

[3 File Finding Design 7](#_Toc357176644)

[3.1 Overview 7](#_Toc357176645)

[3.2 File searching 7](#_Toc357176646)

[3.3 Finding Runs 8](#_Toc357176647)

[4 Python 9](#_Toc357176648)

[4.1 Exports 9](#_Toc357176649)

[5 High-Level Interface 9](#_Toc357176650)

[5.1 Searching Properties-File Defined Directories 9](#_Toc357176651)

[5.2 Legacy FileFinder Object 9](#_Toc357176652)

# Introduction

## Purpose of this Document

This document describes the design of the system for file searching within Mantid. The current implementation started life as a simple check of a list of directories for a given text string but has since ballooned to include things such as querying remote archives, handling wildcards and case insensitivity on case-sensitive file systems.

The new design, as documented here, will consider the above requirements (and others) from the start.

## Scope of this Document

This design is intended to cover the activity of file searching within the framework. It will encroach in areas such as InstrumentInfo in a bid to relocate some functionality to its proper place.

## Context of this Issue

This is the first draft of the FFI-DDD. The redesign is considered an internal maintenance task and as therefore will be discussed and approved by the Mantid developers but not with the wider user base.

## Definition of Terms

|  |  |
| --- | --- |
| ADD | The Architectural Design Document, the high level design document for the entire system. |
| URD | The User Requirements Document, records the users’ requirements for the system. |
| SRD | The Software Requirements Document, specifies the behaviour of the software system. |
| API | Application Programming Interface, defines the interface through which two programs may interact. |

# Current File Searching Overview

The current implementation provides a single class with high-level functions for constructing a full path to a given file and attempting to construct a run file from a number and a set of rules. The functions are relatively involved and changing/adding things within them has become a difficult task due to the complexities of the code.

## Python interface

The FileFinder is implemented as a singleton and is exported to Python so it is imperative that the existing methods are deprecated so that users can switch to the new implementations.

Requirements

|  |  |
| --- | --- |
| Specific requirements status flags: | |
| M | Mandatory requirement. This feature must be built into the final product. |
| D | Desirable requirement. This feature should be built into the final product unless its cost is too high. |
| O | Optional requirement. This feature can be built into the final product at the Project Manager's discretion. |
| E | Possible future enhancement. This feature is recorded here so that the idea is not lost. It must not be included in the final product. |
| X | Deleted requirement. Gravestone retained for reference. |

## Requirements for file searching

This is place for storing specific requirements that the file searching system must provide.

| Label | Requirement | Necessity |
| --- | --- | --- |
| P3.1.1 | Given a filename, including an extension, find a matching file given a list of directories (the directories could be remote drives) | M |
| P3.1.2 | Given a run number,an instrument and a list of known extensions: construct a filename and then perform P3.1.1 | M |
| P3.1.3 | Case insensitive matching for case-sensitive file systems | M |
| P3.1.4 | Optionally include a search of the defined list of archives for the current facility | M |
| P3.1.5 | It must be possible to use the searching facilities from Python | M |
| P3.1.6 | Mark specific directories to be searched recursively | D |

# File Finding Design

## Overview

To support the requirements in section 2.2 the following set of classes will be defined:

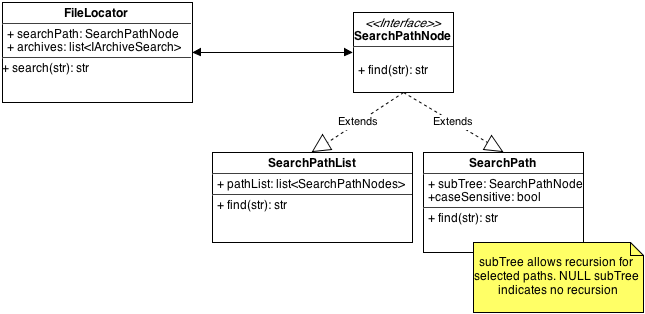
* SearchPath – Defines a single search directory holding a flag to indicate recursive searching
* SearchPathList – Holds a list of SearchPath objects
* SearchPathNode – Interface for SearchPath & SearchPathList, essentially implementing the composite pattern to allow easy implementation of recursion
* FileLocator – Holds a SearchPathNode and will attempt to find a given filename. It knows nothing of instruments, run numbers etc
* RunLocator – An extension of FileLocator that first generates filenames from run numbers and instrument info and then can search using the capabilities of FileLocator.
* Runfile – Encapsulates assembling a run file from constituent parts

The following classes will be extended to support the new class structure:

* InstrumentInfo – A new field, delimiter, will be added that returns the optional string between the instrument and run number. Also a new method, generateRunfiles, will be added that can generate a list of possible Runfile objects given an input string.

## File searching

We will first consider the case of simply trying to find a matching filename within a set of defined search paths. This will require the first four classes defined above where the relationships are show in the class diagram below:



The FileLocator class combines the ability to search over a list of defined paths with the option to search the current archives for a given filename. The archive search will be performed using a stored list of objects implementing the existing IArchiveSearch interface. The string passed to the search method is assumed to be complete and no transformations are applied to it.

The SearchPathNode defines an interface for a node in a tree of defined search paths that is able to take a filename string and return the absolute path if it is found. Two concrete implementations will be provided:

* SearchPath – Represents a single parent directory but with the option to descend into subdirectories. This will allow greater control over which directories are recursively searched;
* SearchPathList – Encapsulates an ordered list of SearchPath objects. The order cannot be changed; a new object must be constructed with the new order.

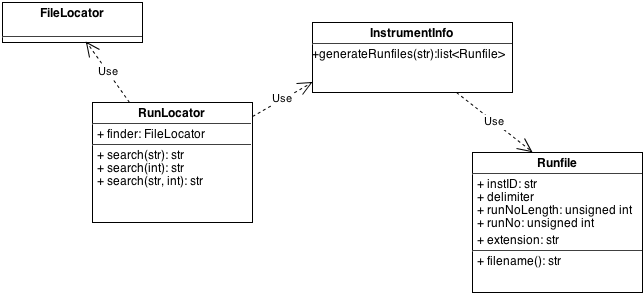
This SearchPathNode, SearchPath and SearchPath classes implement the composite pattern.

## Finding Runs

We are required to be able to find run files from an instrument when given a range of pieces of information from a minimum of the run number up to most of the filename but without the extension. Some examples of expected matching are:

* *PG3\_1234\_event.nxs* 🡪 *PG3\_1234\_event.nxs*
* *PG31234\_event.nxs* 🡪 *PG3\_1234\_event.nxs*
* *1234 with default instrument POWGEN and first extension \_event.nxs*🡪 *PG3\_1234\_event.nxs*
* *SANS2D2000.raw 🡪 SANS2D00002000.raw*
* *LOQ111-add with first extension .nxs* 🡪*LOQ00111-add.nxs*

To facilitate finding runs the following classes will be defined or extended; with the relationships shown in the following diagram.



The RunLocator class processes its input to transform it to a filename or list of filenames before passing them along to its internal FileLocator object to perform searches.

The first job for the RunLocator is to determine which instrument the file will belong to. If no instrument is provided in the string then the current default is used. Once found, the InstrumentInfo object is asked to generate a list of possible Runfile objects that are to be searched for. The RunLocator object then passes filenames generated by the Runfile objects to its FileLocator.

The Runfile object encapsulates the construction of a filename from a series of string elements:

* Instrument identifier – short/long identifier for instrument which may contain numbers, e.g. PG3, SANS2D;
* Delimiter – an optional string separating the instrument identifier from the run number;
* Run number – an integer specifying the run, which can be optionally padded to a given length (specified by the runNumberLength field);
* An optional extension string – this is to be treated more as a general suffix rather than strict file extension to allow only certain combinations of suffix & file extension to appear together, e.g. \_event.nxs.

# Python

## Exports

Each of the additional classes defined above will be exported to Python and be freely constructible by users.

# High-Level Interface

## Searching Properties-File Defined Directories

The current FileFinder searches only those directories defined by the *datasearch.directories* key in the ConfigService. The above framework is completely transparent to this key because the mechanism for locating a file should not be tightly bound to the set of directories that are searched. For example, the above mechanism could be extended slightly to search for files with a given extension and be used by the LibraryManager when searching for plugins.

However, the most common activity with regards to files is searching for data files in the directories defined by the above key. For this reason a cached copy of a FileLocator object and a RunLocator object will be stored in the FrameworkManager to avoid endless conversions from the string key to the path list every time a user wishes to search the data directories list. Two methods will be added to the FrameworkManager to allow retrieval of these objects:

* runLocator()
* fileLocator()

## Legacy FileFinder Object

### C++

The current FileFinder class will be removed and replaced by calls to the new RunLocator/FileLocator as appropriate. The simple interface provided by FileFinder should be able to be replicated by what will be defined using the new classes.

### Python

The current FileFinder singleton instance is exported to Python under the name *FileFinder.* This will first be deprecated to allow users to switch their scripts and the calls routed to the new classes. After a few releases it can be removed.

**ISSUE: Is this a good idea? Should be just keep FileFinder and have it be the one that stores the RunLocator/FileLocator objects that search the** *datasearch.directories* ***key?***

# Multiple Files

**TO DO**