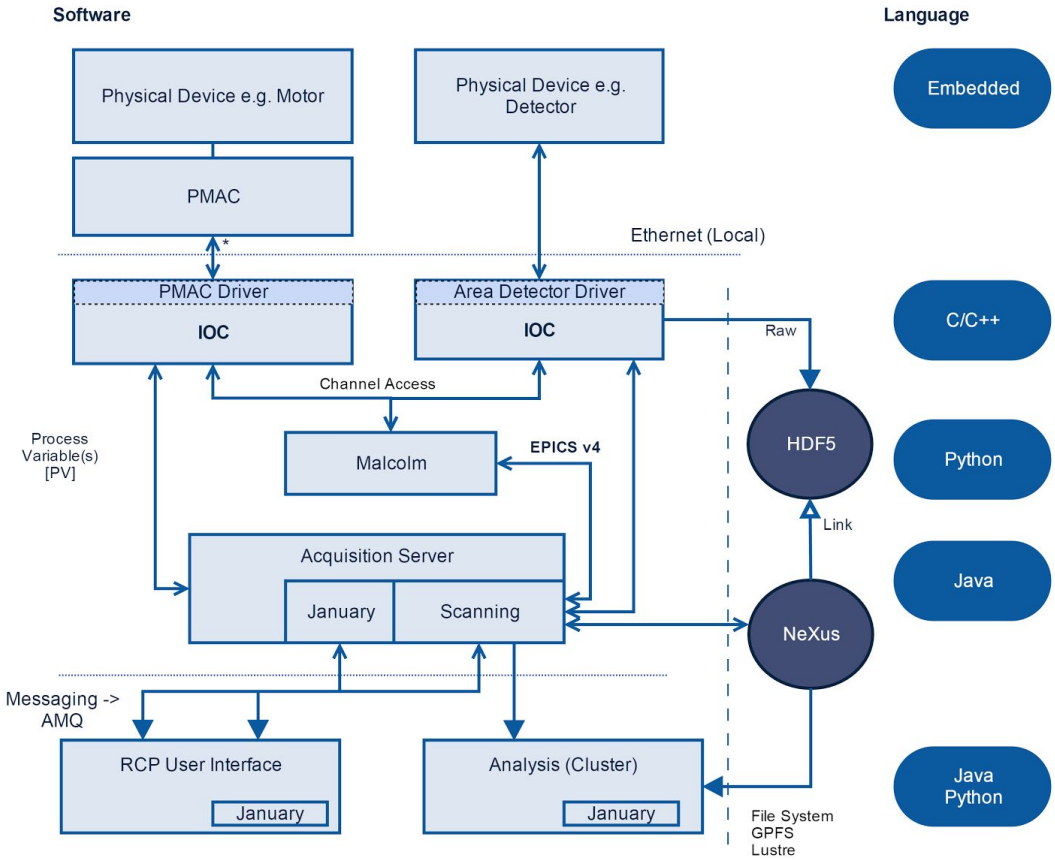
# Anatomy of a Scan in Eclipse Scanning™

## Introduction

Scanning is an open source project for moving scientific instruments and writing NeXus (http://www.nexusformat.org/) compliant files. It is designed to be control system neutral, EPICS, TANGO etc. may be used. See https://projects.eclipse.org/proposals/scanning and https://github.com/eclipse/scanning/blob/master/GETTINGSTARTED.pdf

In this document, what we want to show you is how to run scans and what running a scan is. Firstly we should really say running the default scan because the scanning system is pluggable and different algorithms may be used. However assuming that you would not like to go to the effort of creating new scanning, then this document explains how your devices run with the project. Eclipse Scanning is integrated into GDA9-Solstice available on several beamlines at Diamond Light Source. This document assumes some knowledge of your systems that reuse scanning. For instance at Diamond Light Source the scanning is incorporated into the Acquisition Server (GDA9-Solstice) as follows:




## The Players

### IScannable

The first of the players are IScannables also known as just ‘scannables’ or sometimes ‘axes’ when writing a NeXus file. They are created using Spring (usually) or Jython, or any way you prefer in your scanning server. They connect to EPICS or TANGO devices; they might be motors, for instance, or a temperature controller. In a scan over stages “x” and “y”, both x and y are instances of scannables. The scannable does two things, it manages (sets and gets) the device value and it writes the value to NeXus using a well defined API (see INexusDevice). Scannables have a level and may be positioned using an IPositioner. The IPositioner moves scannables of a given level asynchronously. This provides a basic collision avoidance mechanism because scannables can be set to never move at the same time. (More complex collision avoidance is usually already dealt with at the control layer.)

Scannables are managed by the OSGi service IScannableDeviceService. This service allows scannables to be registered and retrieved by name.

Scannables may contain methods annotated with any annotation defined in Device Annotations . These methods will be notified at different points of the scan. See Annotated Devices below.

Scannables may either be a moveable in the scan or a monitor run as part of the scan. The difference is that a moveable scannable is ‘moved’ to a set-point during the scan whereas a monitor has a read only done on its value and is written to the file. There are different types of monitor, those written at the start of the scan and those written at every point.

### IRunnableDevice and IWritableDetector

The second players in the scan are devices which run with the scan to write data from detectors or instruments or other kinds of read output such as ion chambers. Example of these devices are IRunnableDevice and IWritableDetector in the code and we call them ‘runnable devices’ in English. Runnable devices are not positioned like a scannable but they have a run() and often a write() method, more description about when these are called is below in Scan Timing. The devices also write NeXus and implement INexusDevice however they are writing larger blocks of data, usually an image at each point. The runnable devices deal with how the underlying control mechanism drives the detector. One example of this is EPICS area detector and another is a PyMalcolm Device.

Runnable devices are managed by the OSGi service IRunnableDeviceService. This service allows detectors to be registered and retrieved by name.

Runnable devices may contain methods annotated with any annotation defined in Device Annotations . These methods will be notified at different points of the scan. See Annotated Devices below.

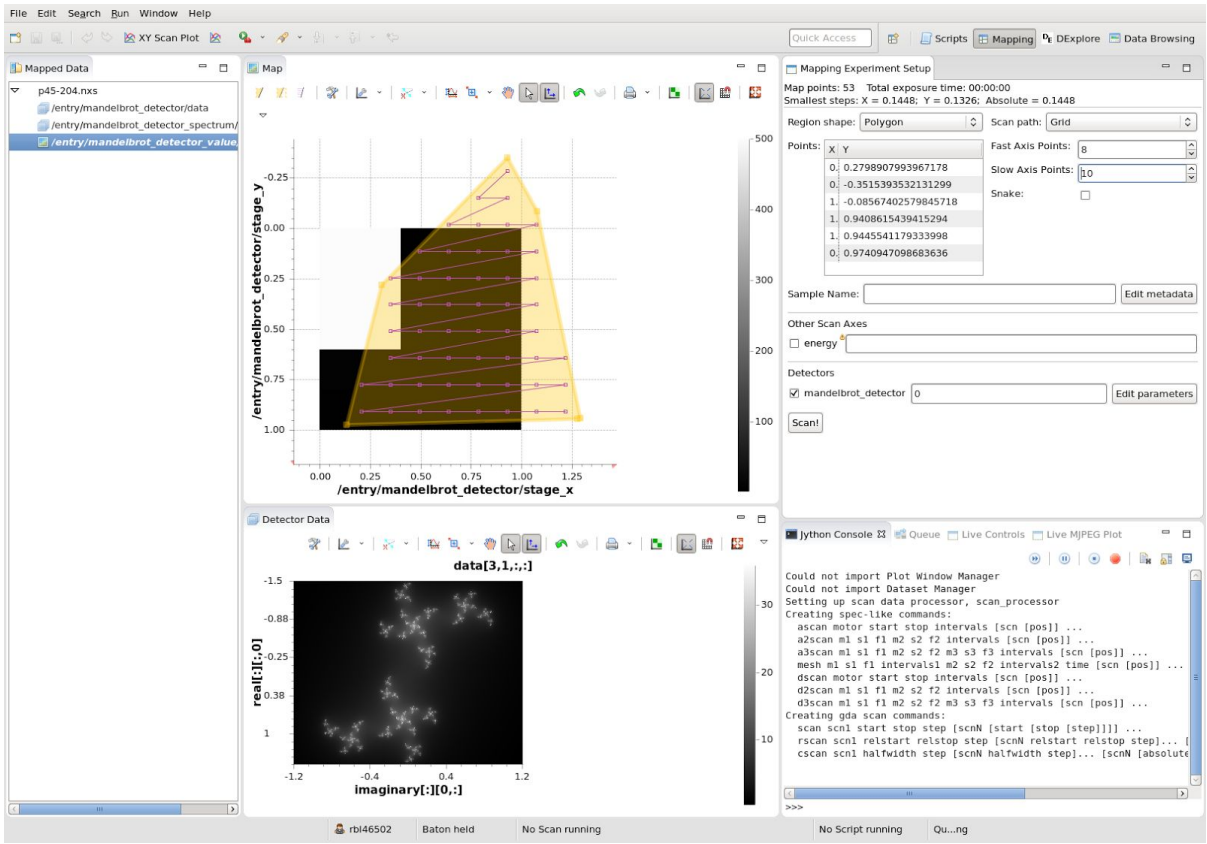
### Annotated Devices

The final player in the scan is the annotated device. This is any object which would like to participate in the scan which does not control one of the above devices. It is an object which may contain methods annotated with any annotation defined in Device Annotations . These methods will be notified at different points of the scan. Scannables and Runnable devices are automatically annotated devices when included in any scan, even if they have no methods annotated.

## Scan Paths

Scan paths are a series of nD positions each of which is reached during the scan using an IPositioner. The paths are managed by a service called IPointGeneratorService which allows paths to be created from simple models. The path follows a Python generator pattern or in Java an Iterator pattern. So an object is produced from the service which provides a series of motor positions. This series of positions is what the scan uses to move to each location. Each location is encapsulated into a single IPosition object. This holds the demand value of each motor at that point and it also holds the location in the data which should be written (which is also known as the data indices).

Scan paths support unlimited regions of interest. It is possible for instance to create a grid scan (a grid of positions or a list of two dimensional positions) but then cut out a polygon from the grid and only scan those values contained by the polygon. Many different shapes of region are supported see ROIs.




Scan paths may be created by nesting other paths so for instance a two-dimensional grid may be surrounded by a step scan. This would give an iterator (or generator) of three-dimensional locations to which the scan will move.

Scan paths are driven by simple models which are easily edited and nested together (we use the term ‘compounding’) to produce complete scans.

Scan paths are interpreted by the IPointGeneratorService for models via a python layer. This is done so that any python programmer may add their own scan paths and models. For instance they might wish to interact with current hardware state when generating the list of points. Choosing Python as the language which generates the points allows anyone to extend that system on-the-fly.

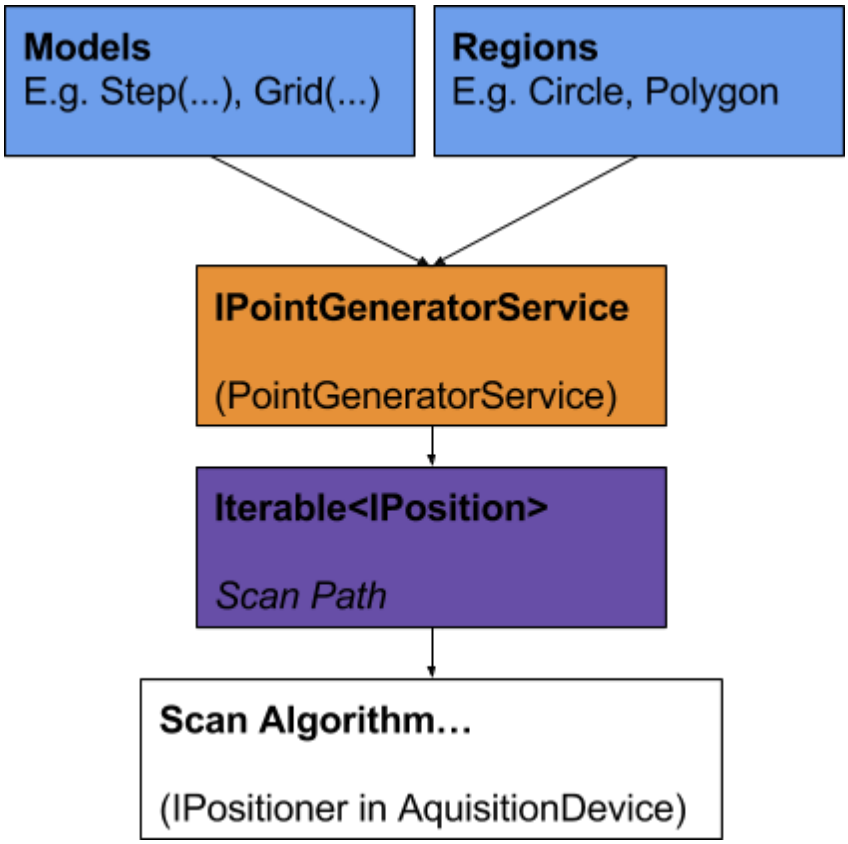
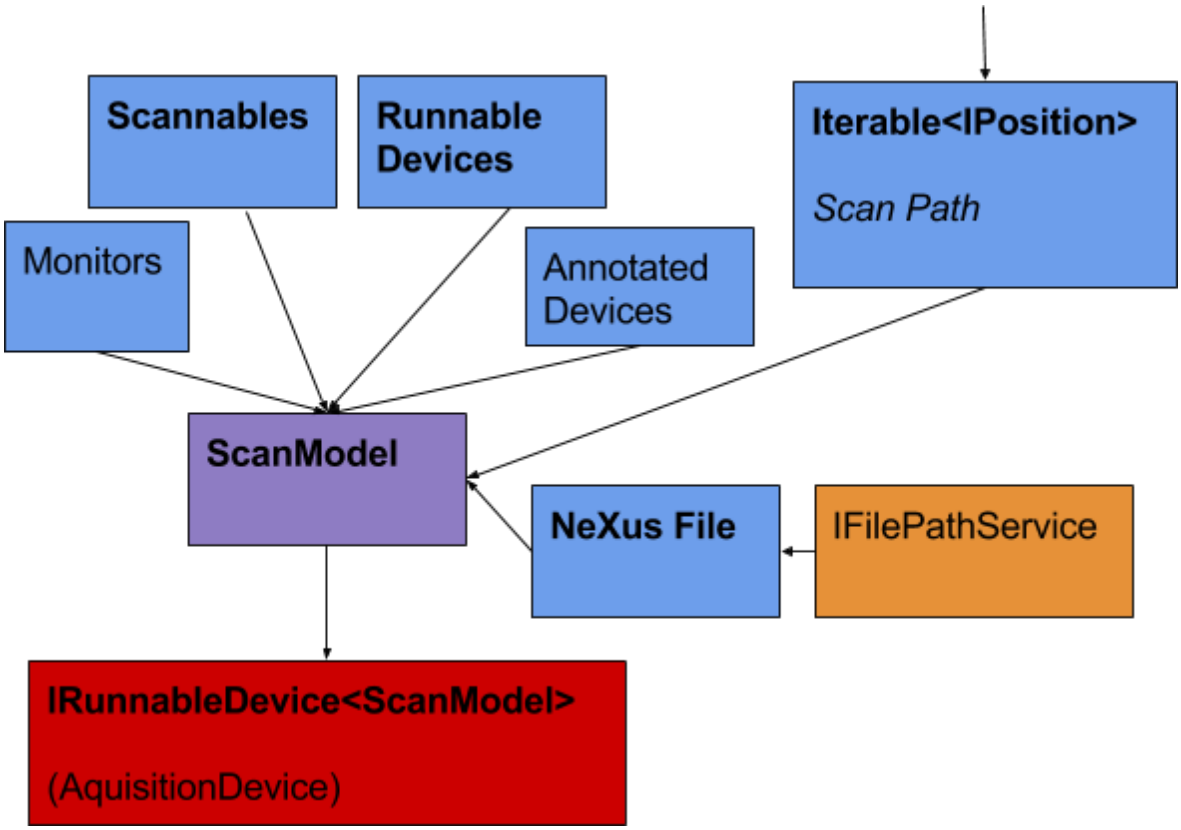



**Figure 1 - Making a Scan Path**

# Scans

## Running a Scan

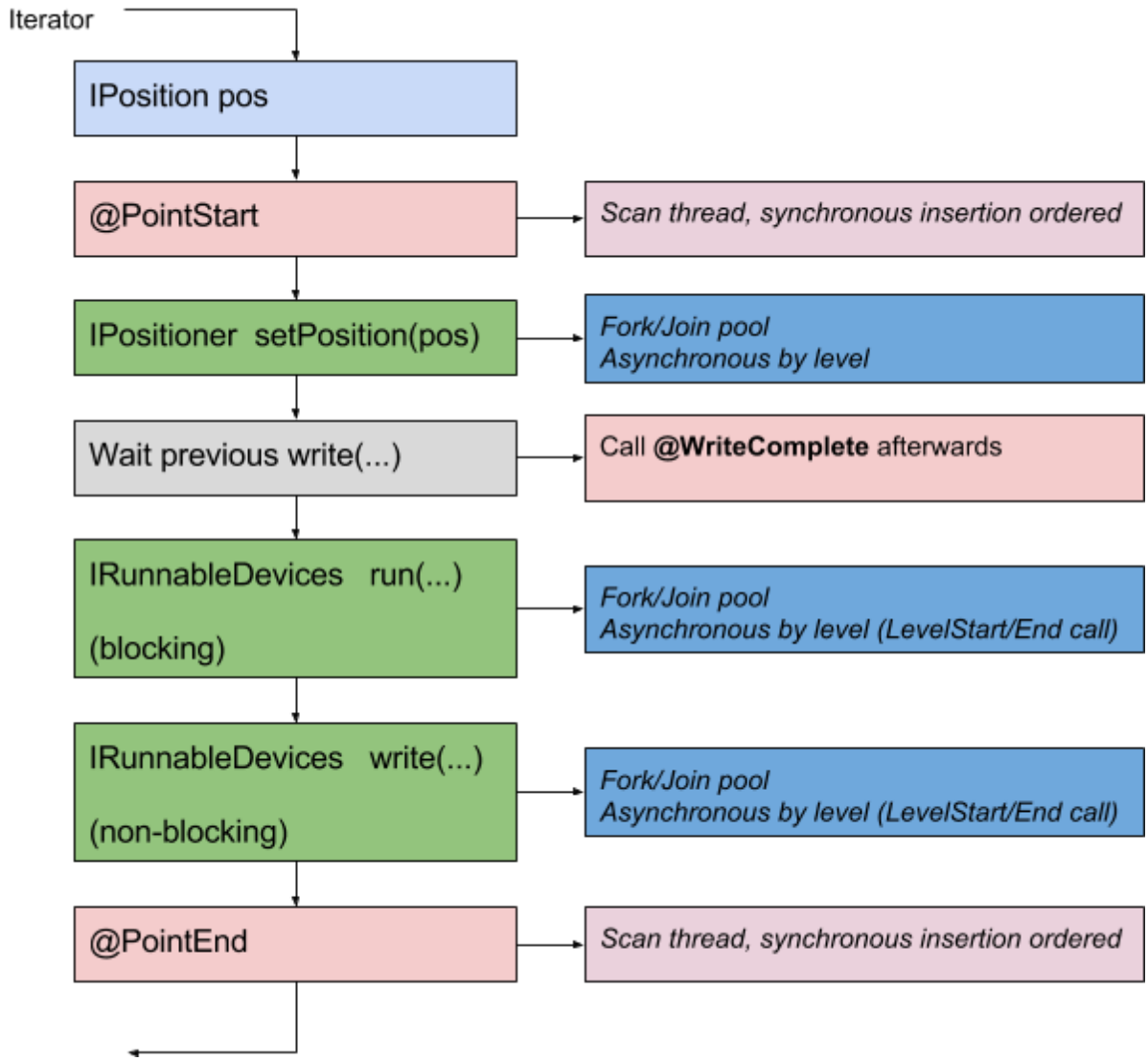
The algorithm which orchestrates the core of the scanning is pluggable and the default implementation is AcquisitionDevice. One point of interest is that acquisition device is also a runnable device, it has a model and a run() method. This means that whole scans may be nested together in principle. However that is not recommended at this time, instead scan paths should be nested to get complete scans. Whenever the runnable device service is asked for a runnable device of model type ScanModel, a device is returned capable of running a scan.




So running a scan is as simple as building an appropriate ScanModel and asking the service for the device to run it. NOTE: When sending a request to run a scan remotely from the client, the object 'ScanRequest' is used. This contains enough information to construct a ScanModel on the server without have access to all the devices.

## Scan Timing

So how does the scan algorithm actually proceed? Simply put it iterates the positions, moves to them then calls the run() method on all runnable devices. Once the run() method returns, the write() method is called and the next position is seeked at the same time. This allows write() methods to be dealt with (e.g. getting data to disk) while the next position is being moved. There is a diagram explaining this process below and the code is in the easily understandable AquisitionDevice.run(...).




## Conclusion

Scanning is a flexible multi-threaded framework designed to be easily supportable for major facilities. It incorporates some of the popular concepts in the GDA framework but releases them in a product neutral open source repository with the aim to foster collaboration between different facilities.

If you have any comments or suggestions about extending the scanning or creating your own please raise an issue at https://github.com/eclipse/scanning or send an email to scanning-dev@eclipse.org.