# Table of Contents

Herschel Data Processing: Architecture & Design ............................................. 1

Introduction ........................................................................................................... 2

Purpose ................................................................................................................... 2

Audience .................................................................................................................. 2

Design Considerations .......................................................................................... 2

Design Considerations ......................................................................................... 2

Elaboration of Risks ............................................................................................... 3

References .............................................................................................................. 3

Applicable Documents ......................................................................................... 3

Conceptual Breakdown .......................................................................................... 4

Within the Herschel Science Ground Segment .................................................... 4

As part of the Data Flow ......................................................................................... 4

Applications Overview .......................................................................................... 5

Main Applications ................................................................................................. 6

DP Libraries .......................................................................................................... 7

HCSS Library ......................................................................................................... 7

Functionality Overview ......................................................................................... 7

Main Data components ......................................................................................... 7

Main Functionality components ......................................................................... 8

Components External to DP ................................................................................. 9

Services & Use Relationships .............................................................................. 10

Overview .............................................................................................................. 10

Data & Services ..................................................................................................... 10

Data & Processing Services ................................................................................. 11

Data & Storage Services ....................................................................................... 11

Data & Visualization Services ............................................................................. 12

Toolboxes .............................................................................................................. 13

Numeric Toolboxes ............................................................................................... 13

Task Toolboxes ...................................................................................................... 14

Pipeline Toolboxes ............................................................................................... 14

Framework & Structures ..................................................................................... 16

Data Structures ................................................................................................... 17

Numeric Data Structures ...................................................................................... 17

Dataset Structures ............................................................................................... 19

Product Structures ............................................................................................... 20

Storage Framework ............................................................................................. 22

Product Storage API ............................................................................................. 22

Exporting Products ............................................................................................... 26

Processing Framework ......................................................................................... 27

Numeric Framework .............................................................................................. 27

Task Framework ................................................................................................... 29

Pipeline Framework ............................................................................................. 31

Product Generation Framework .......................................................................... 35

Pre-Processing ..................................................................................................... 36

Observer’s Pipeline ............................................................................................... 38

Post-Processing ..................................................................................................... 39

Plugin Design ........................................................................................................ 39

Observation Locator ............................................................................................. 41

Visualization Framework ...................................................................................... 42

General Concepts ................................................................................................. 42

Visualization of Data Structures ......................................................................... 42

GUI Application Elements ................................................................................. 43

Interpreter Framework ......................................................................................... 49

Interpreter advantages ......................................................................................... 49

Jython Interpreter ................................................................................................. 49

Command Line Interface ....................................................................................... 49

Documentation Framework .................................................................................. 51

End-user Documentation ....................................................................................... 51

Developer Documentation ..................................................................................... 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observation Data Hierarchy</td>
<td>53</td>
</tr>
<tr>
<td>Observation Context</td>
<td>53</td>
</tr>
<tr>
<td>As User API</td>
<td>53</td>
</tr>
<tr>
<td>A State Machine</td>
<td>53</td>
</tr>
<tr>
<td>ObservationContext Design</td>
<td>54</td>
</tr>
<tr>
<td>Telemetry Context</td>
<td>55</td>
</tr>
<tr>
<td>Auxiliary Context</td>
<td>56</td>
</tr>
<tr>
<td>Calibration Context</td>
<td>57</td>
</tr>
<tr>
<td>LevelX Context</td>
<td>57</td>
</tr>
<tr>
<td>Quality Context</td>
<td>57</td>
</tr>
<tr>
<td>Applications</td>
<td>58</td>
</tr>
<tr>
<td>Observer Interactive Analysis</td>
<td>59</td>
</tr>
<tr>
<td>Calibration Interactive Analysis</td>
<td>60</td>
</tr>
<tr>
<td>Environment</td>
<td>60</td>
</tr>
<tr>
<td>Dataflow</td>
<td>60</td>
</tr>
<tr>
<td>Systematic Product Generation</td>
<td>61</td>
</tr>
<tr>
<td>Storage Configuration</td>
<td>61</td>
</tr>
<tr>
<td>Plugin Configuration</td>
<td>61</td>
</tr>
<tr>
<td>Pipeline Configuration</td>
<td>61</td>
</tr>
<tr>
<td>On-demand Product Generation</td>
<td>62</td>
</tr>
<tr>
<td>User Front End</td>
<td>62</td>
</tr>
<tr>
<td>Backend Engine</td>
<td>63</td>
</tr>
<tr>
<td>Bulk Product generation</td>
<td>65</td>
</tr>
<tr>
<td>Quick Look Analysis</td>
<td>66</td>
</tr>
</tbody>
</table>
Herschel Data Processing: Architecture & Design

This document describes the Architecture and Design of the Herschel Data Processing system.

**Document ID:** HERSCHEL-HSC-DOC-0957

**Issue:** 1.0, 5 August 2008

**Approved by:**
- Jorgo Bakker (ESA)
- Steve Guest (SPIRE)
- Rik Huygen (PACS)
- Peer Zaal (HIFI)

---

**Introduction**

Guidelines and policies for this document are laid out here.

**Conceptual Breakdown**

Describes how the Data Processing Architecture fits into the Herschel Ground Segment and defines the major modules in the Data Processing System.

**Services & Use Relationships**

Describes the services provided by the Data Processing Architecture and presents the relationships among them.

**Framework & Structures**

A description of the major software frameworks and structures upon which the Data Processing System is built.

**Observation Data Hierarchy**

Describes the design and usage of the Observation Context and other data associated with it. This hierarchy is a management structure for the data within an observation.

**Applications**

This chapter describes the major applications within the Herschel Data Processing software.
Introduction

Purpose

This document provides a top-level view of the architecture of Herschel Data Processing (DP) System. It describes where DP fits into the Herschel Science Ground Segment as well as the relations between the various sub-systems of DP. It describes the architecture and major components of each of those systems, but does not provide a detailed description of the design of those major components. This is left to the detailed design documentation provided in the package documentation of each individual component.

Audience

The main audience of this architecture and design document are the architects and developers of the system. However, software testers and software managers may find this document helpful in understanding:

- the relations between all software components,
- what needs to be tested and
- what still needs to be developed.

Design Considerations

Development a system is always bounded by some restrictions. This section is giving a brief overview of these restrictions and how they impact on the architecture and design of the DP system.

Design Considerations

Before any development can start, first two fundamental questions need to be answered:

What to build

What kind of software do we have to provide, to whom do we have to provide that software, when is it needed, is it stand-alone development or part of a bigger whole, etcetera...?

This software is primarily intended for the data processing of the Herschel downlink data. It will be used in operational environment, post-mission environment for archiving purposes as well as on the astronomers computer to further reduce the data, whilst it must also support the instrument teams during the development of the instrumental hardware.

As such it will be developed as an integral part of the Herschel Ground Segment Software while maintaining the possibility to run the data-processing in a non-operational environment such as an astronomer’s laptop.

To meet the latter requirement, it is highly desirable that those components of the data-processing software are sufficiently decoupled from the operational specific components. Moreover, it should be able to install and run that part of the software on popular operating systems. In addition the required software components should be license free.

How to build

What infrastructure we are going to use, what development languages will be used, how do we keep control over source code configuration, who is going to do what?

To allow full integration with the rest of the Herschel Ground Segment software, the Java language is adopted. Jython is chosen as a scripting language, for its access to Java components and its powerful abbreviated syntax features.

The source code configuration and automatic daily building techniques are adopted from the Herschel Common Science Software (HCSS) system. As such the Data Processing software is built as an integral part of the HCSS system.
Elaboration of Risks

It should be noted that decisions made in the elaboration phase are mainly driven by potential risks encountered during the development of the Herschel DP software:

Requirement risks

Do the requirements provide sufficient insight in what is required to develop such a system. Are we building the wrong system?

The user requirements are captured in the HCSS URD [AD1] and further elaborated in the form of use cases [AD1] and supplementary specifications [AD2].

Technological risks

Are the technologies that we use available and stable? Will they be available during the lifetime of the project, including the time that customers are using the software? This applies to languages, repository systems, documentation systems, testing facilities, hardware platform specifics and so on.

Skill risks

Do we have the skills in house? For example, is a particular technology deployable given the skills that are available to develop and maintain the software?

The development team for data processing software is a mix of developers with different backgrounds in software development as well as different levels of knowledge of the software languages used within the project. This includes the understanding of the needs of test harnesses, documentation of software interfaces and so on. It is best that these risks are encapsulated by providing a pluggable nature to the system, such that potential problems can be quickly isolated and repaired. On the other hand, a number of developers have astronomical background which is considered an advantage because they better understand the mechanisms used for data processing and the requirements of the users.

Political risks

Are there political forces that may affect the software project as a whole? Do these forces influence the way the project is financed?

Geographical risks

How and where are the developers located? How far are they away from their customers? What communication channels can be provided?

Developer teams and users are geographically scattered around the world and this can be considered as a high risk. The same applies for the customers of this software. Fortunately, the risk is somewhat lessened by the fact that the distribution of developers and customers are generally such that on almost each site a concentration of multiple customers and developers can be found. Unfortunately, these developers tend to be highly oriented towards the customers at hand, and not the customers in general. This tends to results in so-called single-user satisfaction.

References

Applicable Documents

- [AD1]: HCSS User requirements Document, FIRST/FSC/DOC/0115
- [AD2]: HCSS Data Processing Use Case Definitions, HERSCHEL-HSC-DOC-0480
- [AD3]: HCSS Data Processing Supplementary Specification, HERSCHEL-HSC-DOC-0481
Conceptual Breakdown

Within the Herschel Science Ground Segment

Mission data is organized around observations. Observations are represented in the Core Class Model (CCM) that resides in the object database. The database, along with the protocol (abstraction layer) used by software to communicate with it, is shown at the center of the following figure.

Proposals are made up of observations. Each observation is turned into several sets of commands by the Common Uplink System (CUS); and scheduled at some time by the Mission Planning subsystem. All of this work is captured in the CCM and relationships among data object about an observation are associated by a common obsid, or observation ID. The uplink commands for an observation, organized by obsid, are placed in a telecommand package and radiated to the observatory, where the obsid is used to identify data packets radiated back as telemetry. The Telemetry Processor stores telemetry in the CCM, so downlink data becomes associated by obsid with uplink data there, in the CCM. As telemetry is used to make data frames, and data frames make products, the obsid is carried forward, maintaining the relationships.

The common view of telecommanding packages, telemetry, and science data comes from the data objects stored in the object database, and organized by object ID.

As part of the Data Flow

The following diagram shows an overview of the Data Processing data flow.

Telemetry and auxiliary information arrive from the MOC to the Herschel Science Centre (HSC), with the aid of the File Transfer System (FTS) and the Data Distribution System (DDS). On one hand, telemetry is converted to data frames, and stored, in both processed and raw forms, in the HSC Object Database (HOD), which implements the CCM. On the other hand, auxiliary information is processed and stored in the form of Auxiliary Products, in the Herschel Science Archive (HSA), which is backed by a relational database.

The ICCs elaborate and provide Calibration Products to the HSC. These products are left in a temporary area, from where they are ingested into the HSA following a manual procedure, driven by a Change Control Board (CCB) that monitors and validates them.
The **Systematic Product Generation** (SPG) is in charge of generating Level X Products and Quality Products, through what are called the **pipelines**, which are specific per instrument. Before running a pipeline, there is a pre-processing phase, built up by plugins, which collects data frames as well as raw telemetry if needed, for generating Level 0 Products. The output of the pre-processing phase is an Observation Context associated to the actual observation, which points to the relevant Auxiliary Products, Calibration Products, and the generated Level 0 Products. The Observation Context in this stage is the entry point for the pipelines, which can be run outside the HSC, e.g. in an ICC or an astronomer’s laptop, since a connection to the HOD is not needed for the processing, and the HSA can be replaced by a PAL storage within the pipelines. Finally, Quality Products are generated in the post-processing phase, which is built up by plugins again.

An ICC or astronomer can connect to the HSA to browse and retrieve the available products. This may be done through the **Herschel Archive User Interface** (HUI), or by means of the **Product Access Layer** (PAL). Both mechanisms can be used from within the IA application, the **Herschel Interactive Processing Environment** (HIPE); the HUI can be executed from a dedicated web server as well.

### Applications Overview

The Herschel DP software is built up as follows:
Main Applications

The main applications can be broken up into the following groups:

Calibration Interactive Analysis (CIA)
Note that this also includes the Trend Analysis (TA) system. Essentially all functionality provided in this application are to improve calibration data and to improve algorithms applied in the processing software.

Observers Interactive Analysis (OIA)
The Observers Interactive Analysis system provides the functionality to reduce the raw data (also referred to as Level0 Products) using the same modules as facilitated within operational environment as well as functionality to access the data found in the Herschel Archive, to visualize it and to store it locally for further processing and offline processing.

Systematic Product Generation (SPG)
The Systematic Product Generation system provides the functionality to systematically collect relevant data; to process that data from its rawest form down to level0; to process that raw data to LevelN products using the same modules as will be provided to the astronomer with the use of default control parameters; production of quality data, logging and history data; the quality inspection and ingestion of reduced data back into the Herschel Archive.

Bulk Product Generation (BPG)
This is a flavor of the Product Generation system, where many observations need to be re-processed. Typical reasons for bulk re-processing are fundamental bugs in reduction algorithms, identification wrong calibration data or new insights in calibration data, missing data in the output (e.g. required for quality and trend analysis).

On-demand Product Generation (OPG)
This flavor of the Product Generation system allows astronomers to re-process data with a different set of control parameters than used by the SPG system. Though this kind of re-processing can be done within OIA, OPG provides a service for those who can or will not install OIA on their own computer systems.

Trend Analysis (TA)
Trend Analysis in se is not an application by itself but uses all the functionality provided by data processing to analyse data values over a long period of time, up to the whole mission. The goal is to look for possible trends in both housekeeping or science data or find unknown dependencies.

Quality Control Browser (QCB)
The Quality Control Browser application provides to the Scientific Product Analyst, and to any other actor involved into the Quality Control Pipeline, a graphical tool to view the information contained into the quality report linked to any observation already processed.
Quick Look Analysis (QLA)
It provides the streaming analysis functionality typically used by instrument engineers and calibration scientists during the development of the hardware to quickly identify issues in the data produced by the hardware.

**DP Libraries**
The DP libraries comprise of the following major components:

- **DP Core component**
  - This component provides library software that is shared across all instruments and applications.
- **DP HIFI component**
  - This component provides library software specific to the HIFI instrument
- **DP PACS component**
  - This component provides library software specific to the PACS instrument
- **DP SPIRE component**
  - This component provides library software specific to the SPIRE instrument

**HCSS Library**
The HCSS library is developed jointly by the Herschel Common Science Software team and provides modules that are used within the Herschel DP software.

**Functionality Overview**
The functionality of Data Processing is centered around the data products. In other words, the contents, shape and size of the data dictate the shape of the functionality services that are provided.

The main reason for this is that the data volume can be quite large; for example, the data required for a particular computation necessitates the algorithm to be written such that it can be processed within the computing environment.

**Main Data components**
Processing Herschel data requires several data types, which can be grouped into the following areas:
Raw Products
The raw data, as contained within the science telemetry of observations is the prime input data.

Calibration Products
This type of data is required to calibrate the raw data coming from the various Herschel instruments.

Auxiliary Products
The auxiliary data provides information about the pointing of the spacecraft, out-of-limit information and so on.

Standard Products
The various levels of processing produce results, which will be provided in the form of standard products; that is, Products are generated from the systematic processing of the Raw Products with standardized processing environments. An Astronomer can still decide to regenerate these products with non-standard processing environments such as adaption of processing control parameters or even replacements of processing modules.

Quality Products
The results of a processing run are inspected for their quality. The findings of such inspection end up in a Product which is called a Quality Product.

It is desirable to provide similar interfaces to interact with these Products. That is, learning to interact with one type of Products gives you insight to interact with all the others.

**Main Functionality components**

The Data processing system provides all sorts of facilities to process the data. The functionality can be roughly grouped as follows:

Command-line
Users want to be able to interactively execute processing commands and to be able to query and inspect their data from that command-line.

Scriptable Pipelines
It should be possible to chain processing tasks with logic to connect them in a scriptable environment; that is, no re-compilation of source code should be required when the logica and processing steps within a pipeline is changed by the user of the system. Likewise, users should be able to create processing scripts themselves. This environment shall provide the same look and feel for the pipelines across all instruments.

Pipeline Tasks
A user should be able to create new tasks in a scriptable language so that these tasks can be loaded and used as a single entity or as part of a new or modified pipeline of tasks. It is highly desirable to provide a standardize way of developing tasks, such that the interaction with them has the the same look and feel across all tasks.

Numeric Algorithms
A user should be able to apply pre-defined, well performing numeric algorithms on their data arrays either directly from the command-line or as part of their tasks. It is highly desirable to provide a standardize way of interacting with such numeric algorithms.

Product Browser
A user should be able to browse through all available Products within their interactive environment; that is, a mechanism to inspect Products that are generated locally by the user himself as well as Products available from the Herschel Archive or any other location that may provide them (such as a site within an Instrument Consortium).

Archive IO
A user of our system should be able to access the data available within the Herschel Archive. This access is subject to access rights, of course. An astronomer may have only read access to public data; a calibration scientist may have read access to everything in the archive and an operator of the standard processing in the operational environment may have write access as well.

Local IO
A user should be able to have read and write access to locally generated data. The interfaces used to access that data shall be similar or even the same than when they access archived data, to avoid unnecessary amount of interfaces to get data from different sources. It should not be forgotten that astronomers like to work with FITS files, so means to export data to FITS files are mandatory.

Help & Documentation
Documentation is provided on all levels required. This documentation should be geared towards the type of user that is interacting with the system. A user that is only interested to re-run pipeline processing with different parameters should be provided documentation on how to access data and how to interact with a pipeline.

Herschel Data Processing Architecture & Design, Issue 1.0
Users that want to execute pipeline tasks separately in e.g. batch mode should be provided documentation on how to interact with those tasks in question and how to write loops and all that within the provided scripting environment.

Users that develop their own tasks should be provided documentation on how to write these tasks, what facilities they may use to build up their algorithms and how to plug these newly developed tasks into their environment.

For lower level developments, a user should have access to developers documentation such as Application Programmer Interfaces (API) documentation for the languages at hand.

**Components External to DP**

Apart from the components described in the document, components within the HCSS library will be used by the data processing environment as well:

**HCSS Components shared by all Processing Applications**

These components will be provided to the community and shipped with Observer’s IA. A typical example is the HCSS *share* module.

**HCSS Components shared by Instrument Teams**

These components will be shipped with Calibration IA. A typical example is the HCSS *store* module that implements two-tier access to an object-oriented licensed database.

**Components specific to Operational Environment**

These components, such as the business logic of ESA to ingest and archive Products are developed by the Herschel Science Archive Team (HSAT) and are proprietary to ESA. The client software to access to archive within Observer’s IA is of course provided to the community.

**Components specific to all Instrument Teams**

These components aid these teams in the early phases of hardware development and are proprietary to the instrument teams.
Services & Use Relationships

In order to understand the kind of architecture required for data processing, it is useful to have a kind of high-level overview of services that are required and the way these services depend on each other. Such an overview is void of details, but represents dependencies between various services in the data processing environment. Note that the dashed arrows describe a use relationship.

Not all relationships may be presented, but the most important ones: the diagrams should be just good enough to grasp the relationships between the various services.

Overview

A high level overview of the services within the Data Processing are depicted in the diagram below.

The front-end for most of the users of our data processing system is the Console. It provides access to an interpreter, processing services (pipelines, tasks, numeric algorithms) and visualization services.

Visualization of data and processing of data require access to storage services. It should also be mentioned that the data -though being a fundamental part of the data processing- is not considered a service in this diagram, but inherently the glue between various services such as the services mentioned above.

For operator purposes of systematic processing or bulk re-processing of the observational data, scheduling services will kick off the processing services and one thing and another need to be monitored (is everything going OK? How far are we in processing an observation?).

Data & Services

As mentioned before, the data is been left out of the picture; however the data provides the fundamental glue between the main services provided, which are:

- Processing Services,
- Storage Services and
- Visualisation Services

The sections below discuss these relations in more detail.
Data & Processing Services

Processing or computation services are centered around data granularity:

- **Numeric toolboxes**
  - A set of numeric toolboxes provide the implementations of Numeric functions that can operate on Numeric Arrays.
  - These functions are grouped in toolboxes; that is, all interpolation functions should go into an interpolator toolbox.
  - The implementation of these low-level functions should be done such that the user can expect a performant system.
  - To ensure similar look-and-feel across all these functions, a toolbox framework is provided.

- **Task toolboxes**
  - Tasks are functional entities that operate on collections of data, usually grouped into a Product or a Dataset.
  - These tasks can also take care of the units associated to the data, provide additional meta-data and -in the case of Products- history (how did this task generate its output).
  - The task framework ensures that the behavior of all tasks implementations is similar.

- **Pipeline toolboxes**
  - Pipelines are a collection of specialised tasks organised in a script with a minimum of logic to string these tasks together.
  - The specialisation of tasks lies in the fact that they add history to their output. Therefore they take Products as input and produce Products as output and in addition pipeline tasks can be configured with a set of simple parameters as well. It is envisaged that for each instrument and each observing mode at least one pipeline will exist.
  - The pipeline framework ensures that all pipelines are using similar facilities ensuring that the user can quickly interpret the logic of such a pipeline.

Data & Storage Services

The main storage strategy is centered around products. The Product Storage framework provides a unified interface to store, retrieve and query Products, whether these products are stored locally or remotely (e.g. a product archive).

In order to ensure the relations between the potentially large amount of products that are generated as well as used (as input) by e.g. a pipeline, this framework also provides functionality to maintain these relations.
The storage services provide a means to create different implementations of this storage functionality, e.g.:

Local store implementations
These implementations provide the possibility to communicate with products generated on e.g. a local laptop. An implementation can also provide means to import an existing set of Products into the Data Processing system. Such an implementation can typically perform queries quite well on a small set of products (say a thousand).

Two-tier
Two tier implementations can store products with the use of relational or object-oriented databases. These implementations are geared towards query performance and storage of large amounts of products (say millions).

Three-tier
By providing an additional tier, we can provide access to the above implementations to the external world. That is, wrapping a local store into a three-tier solution an astronomer can share the locally stored data with other colleagues and the archive can provide access to automatically generated and archived data to remote users.

Caching
This implementation provides the means to cache remotely accessed Products on the local system. This opens up the possibility to enhance retrieval performance for those remote products that are accessed multiple times on a remote system as well as the opportunity to work off-line with those products already retrieved.

The interpreter is used to ensure that the query syntax is the same as the syntax used on the command-line whenever a user communicates with a product, and the Product Browser uses the Product Storage to visualize the data stored in all the configured pools.

**Data & Visualization Services**

Visualization of the data is broken up along the same granularity as the data entities provided by the DP system.
On the numeric level facilities are created to generate XY plots, 2D images, surface plots, and cube visualization. The Product Explorer framework provides the means to create Product and dataset visualizations which are geared towards presenting the data contained within such a product or dataset by combining the lower level facilities into a coherent graphical look on that data. One can think of providing a explorer that visualizes a spectrum product by populating an XY plot, setting the axes and titles, and listing accompanying meta-data.

When accessing stored products, the Product browser uses the Product Explorer framework to visualize the retrieved Products.

**Toolboxes**

The computational services are provided and organized in the form of toolboxes. Though not shown here, one should keep in mind that certain users will want to create their own computational implementations as well.

This section provides a conceptual layout of these tool boxes; given such a design for the layout of these toolboxes, a user toolbox is easily created.

**Numeric Toolboxes**

The DP system provides a set of numeric toolboxes that hold functionality which is useful to all instruments. One can think of basic functionality such as trigonometry, statistics, integration, interpolation, transformations, matrices, etcetera...

The instrument specific numeric toolboxes hold functionality that can be based on the functionality provided by the common toolboxes (e.g. using the SIN) in order to create higher order numeric algorithms which are instrument specific.
Task Toolboxes

Tasks area dealing with collections of quantified data, combined in products and datasets. Task implementations use a combination of functions found in numeric toolboxes in order to perform a computation specific to all or parts of the data contained in those products and datasets. Likewise, whereas the common tasks are applicable to generic products and datasets (e.g. the help task), the instrument specific tasks perform computations that are specific to the instrument data structures and data contained within those structures.

Below is a simplified view of these toolboxes. The intent is that each instrument organises the instrument specific tasks in categories of functionality.

Pipeline Toolboxes

Though initially a single pipeline exists for each observation mode and instrument, it can be envisaged that in time different pipelines are produced for a single observation mode. E.g. the Observer may take an existing pipeline, modify it and compare the results of the two pipelines. To ensure that the system can deal with a multitude of pipelines and that the Observer can discriminate between pipelines, a need arises to logically group these pipelines. In line with the numeric and task toolboxes, it makes sense to organize these pipelines into toolboxes.
as well.

These pipelines themselves are using mostly instrument specific tasks (taken from the instrument toolboxes) as depicted below:
Framework & Structures

Data processing consists mainly of a number of loosely coupled frameworks and generic data structures. The applications that are part of DP heavily build on these frameworks and data structures. In this section the overall design of each framework is presented in more detail.

Data Framework
Three types of generic data structures are available to applications and to users in interactive sessions. These generic structures can be extended and combined to form more complex and specialized data structures.

Storage Framework
Data storage is mainly centered around Products. The framework provides easy storage (and retrieval) of products on your local system and in remote databases. The interface is independent of the storage mechanism or media that is used, because the storage framework is a layer between those mechanisms and the applications.

Processing Framework
Different levels of processing and computation demand different frameworks for manipulating data structures. Handling numeric data must be both performant and have a small memory footprint. Pipeline processing requires high level functionality for managing the processes and the data products that are involved. In between is the Task framework which provides convenience functionality for developers like automatic handling of history and validation of parameters.

Product Generation Framework
Pipeline processing requires special functionality to prepare and collect input data, schedule and distribute processes over a grid of processors, collect output data and generate reports. All this functionality is isolated in this framework as it is specific to automated product generation.

Visualization Framework
Generic data structures that are extendable need a likely generic visualization framework for inspecting both the structure and the content. This framework provides a plugin architecture for data explorers.

Interpreter Framework
The interpreter framework provides an interactive environment with a command line, a debugging editor and command history.

Documentation Framework
Documentation is as much as possible written in the source code and automatically parsed and extracted into well-organised reference manuals. In addition there are comprehensive manuals written for both the user and developer of this system. This framework provides all the tools needed to have up-to-date manuals with each delivery.
Data Structures

Data structures in Data Processing need to be accessible for a large public. That is, developers have to be able to use them in both lower level code as well as in higher level scripting languages. Users need to be able to work with them and understand them in interactive sessions. Systematic processing scripts need to be able to ingest and to produce these structures whilst attaching history information and the structures need to be such that they can be archived.

To achieve all this, the data framework provides three layers of data structures:

- Numeric Data Structures
- Dataset Structures
- Product Structures

Numeric Data Structures

Structures that hold arrays of simple values, as well as the provision of simple operators. Most of them are in used in the mathematical sense; one has to think of arrays of numbers (discrete, real and complex), but also boolean values.

To ease the access to elements and slices of elements within these arrays, the Java arrays themselves are not sufficient for interactive usage. Though it is possible to write scripting operations -such as a loop- on the Java arrays themselves, not only the performance of these operations can be a bottleneck; the duplication of re-occurring operations can endanger the reliability of the code as well.

In addition we have to try to provide a similar look-and-feel to all operations that can be done on these arrays, regardless the element type and regardless the rank of these arrays.

The numeric data structure framework provide all this functionality. All arrays share the same base interface, and all arrays of a particular elements type share a specialized interface applicable to all ranks. Then the API implementations themselves are geared towards performance and compatibility with storage solutions. The framework and a sub-set of implementations is shown below:
Below some of the properties of array structures are described. Note that list of features or methods mentioned are not complete, but serve as an illustration of the functionality only.

**Generic information**

The ArrayData interface should provide generic information such as:

```java
public interface ArrayData {
    int getRank();
    int getSize();
}
```

**Generic Element Access**

To achieve element access with a syntax on the command-line that is user-friendly, such as:

```java
y = x[i]
y = x[i:j]
```

Additional methods are required, such as:

```java
Object getScalarAt(int longIndex);
ArrayData getArrayAt(IndexIterator it);
Object __getitem__(int longIndex);
Object __getitem__(IndexIterator it);
```

Note: the latter methods are so-called jython hooks and are provided for Jython convenience.

**Type Interfaces**

Specialized type interfaces are available to ensure further commonality across API implementations of arrays of the same type but with different rank as well as giving implementers a chance to write optimized code by avoiding using unnecessary casting. Examples are e.g.:

```java
public interface IntegerArray extends ArrayData {
    int getAt(int longIndex);
    IntegerArray getArrayAt(IndexIterator it);

    IntegerArray setAt(int longIndex, int value);
    IntegerArray setAt(int longIndex, Integer value);
    IntegerArray setAt(IndexIterator it, int value);
    IntegerArray setAt(IndexIterator it, IntegerArray value);
}
```

**Operators**

All array implementations implement certain operators for addition, multiplication, etcetera for performance reasons. In addition, Jython hooks are implemented to provide user-convenience:

```python
# x, y are arrays
y = x + 2  # creation of a new array by adding an array with a scalar
x += 2    # same, but now inline (and therefore using half the memory)
```
This is achieved by imposing interfaces such as:

```java
public interface IntegerArray extends ArrayData {
    IntegerArray add(int rhs);
    Object __iadd__(int rhs); // self+=rhs
    Object __add__(int rhs); // y=self+rhs
}
```

**Dataset Structures**

The above mentioned simple arrays alone are not sufficient; a set of arrays may be related, say in a table, they may need some annotation and accompanying meta-data, and many times the arrays need to be quantified (labeled with a unit) as well. Data set structures provide the means to do so:

**Generic Datasets**

The framework provides three generic datasets:

- **Array Dataset**
  A quantifiable dataset containing array data (say a data vector, array, cube, etc.)

- **Table Dataset**
  A dataset containing a collection of columns. Each column contains a quantifiable array data (say a data vector, array, cube, etc.). All columns have the same number of rows. Together they make up the table.

- **Composite Dataset**
  A dataset containing a collection of named datasets. This allows arbitrary complex structures, as a child dataset within a composite dataset may be a composite dataset itself and so on...

**Specializations**

From the specialized datasets one can build higher order datasets as long as they can be fully specified through the interfaces provided by the generic datasets:
Product Structures

Products are well defined data entities that serve as input as well as output of high level tasks in our system and provide the means to attach history.

Combinations of data sets can be collected into so-called *Leaf* products. Collections of Products can be further grouped by *Context* Products.

Like specializations of Data Sets, one is allowed to create specialized Products as long as they can be represented and specified through either the *Leaf* or the *Context* Product interfaces.
Storage Framework

In section Data & Storage Services, a brief overview of the relations between data structures and the storage mechanism is presented. The aim of the storage framework is to provide a unified storage mechanism of products irrespective of the underlying storage implementation. This way, we can provide a unified access to locally stored products as well as remotely stored products, including querying of those products.

Apart from storing products within the storage framework we also have to provide mechanisms to temporarily store primitive DP data elements (such as array data) -that do not necessarily belong to a product- to the local disk.

The product storage framework is called the Product Access Layer (PAL) and it consists of three main components:

- **Graphical User Interface**: The Product Browser is a graphical interface and serves as a visual front-end for the user. The Product Browser is explained further in the Visualization Framework.

- **Command-line Interface**: The Product Storage is a command-line interface and serves as coding front-end for the user.

- **Plugin Interface**: The Product Pool is a developer interface to provide a mechanism to develop plug-in implementations for the actual storage mechanisms such as local storage, remote storage, caching, etcetera.

The relations between these major components of the PAL can be expressed in a diagram:

Product Storage API

The Product Storage is the core of the PAL and provides the following functionality:

- Registration of multiple Product Pools,
- Lazy loading of Products,
- Lazy saving of Products,
Staged querying of Products,
Versioning of Products,
Tagging of Products

The functionality can be expressed by the following elements:

Product Pool Registry

One can register one or more Product Pools to a Product Storage:

```python
store = ProductStorage()
store.register(localPool1)
store.register(localPool2)
store.register(remotePool1)
store.register(remotePool2)
```

Or in a more straightforward way:

```python
store = ProductStorage(localPool1, localPool2, remotePool1, remotePool2, ...)
```

The first registered pool is always the **writable pool**; the remaining are treated as read-only.

Product Pool Cache

The PAL is able to cache remote pools, for two reasons:

- Avoid unnecessary network load when accessing the same remote Product repeatedly and
- In off-line mode (disconnected from the network), to allow to access those Product that were cached previously

Note that queries are also cached, in such a fashion that on off-line mode, previously executed queries can be replayed.

This feature is not used in on-line mode, as newer Products may have appeared in the remote pool.

The caching follows the Decorator pattern (a.k.a. Wrapper); this is a flexible way alternative to subclassing such that all remote pools can benefit from caching without implementing it themselves though sub-classing:
The resulting command-line may look like:

```java
store.register(CachedPool(remotePool1))
store.register(CachedPool(remotePool2))
```

**Product Identification**

Products are identified by a Uniform Resource Name (**URN**), which uniquely identifies a Product within the set of pools that are registered within a Product Storage setup. Details are discussed in the Design Document of the PAL.

**Product Reference**

A Product Reference provides lazy access to a Product within the Product Storage architecture; that is, amongst others the meta data of a Product. Given a reference, one can access:

```java
ref.urn     # returns the URN
ref.meta    # returns the meta-data
ref.product # returns the full product
```

**Loading & Writing Products**

Given a `urn` variable, one can load a Product Reference from a pre-configured Product Storage:

```java
ref = store.load(urn)
product = ref.product
```

Given a `product` in memory, one can save it to a pre-configured Product Storage:

```java
ref=store.save(product)
```

**Context Products** are light-weight containers of Products; that is, they hold Product references only. The idea is that with Context Products one can provide associations between products whilst maintaining an API which is convenient to the user.

In addition, a Context should know when references to its children are changed, such that saving a Context will also save the changed references. This mechanism provides another level of lazy loading and writing.

**Staged Query**

The PAL can handle queries in a staged fashion:

```java
results = store.select(firstQuery)
results = store.select(secondQuery, results)
```

The results are in the form of a Set<ProductRef>, such that a User can specify for example:
for ref in results.iterator():
    print ref.urn
pass

Product Versioning

The PAL stores information about different versions of the same product. For instance:

```python
p = Product()
p.description = "first version"
ref1 = store.save(p)
p.description = "second version"
ref2 = store.save(p)
p.description = "third version"
ref3 = store.save(p)
```

It is possible to retrieve all versions belonging to the same track, or to get the latest one:

```python
versions = store.getVersions(ref1)  # returns [ref1, ref2, ref3]
lastRef = store.getHead(ref1)      # returns ref3
```

Product Tagging

A product can be named with a tag (label or alias), so it can be accessed quickly later on, without any need to remember its urn or to perform a query:

```python
# Example context pointing to other products
levels = MapContext()
levels.refs.put("level0", level0)
levels.refs.put("level1", level1)
levels.refs.put("level2", level2)
store.saveAs(levels, "levels")

# Later on, it can be accessed through its tag, which works like a urn
levels = store.load("level")

# The urn associated to a tag is available also
urn = store.getUrnFromTag("level")
```

Managing configurations

Creating and configuring a product storage can be a tedious task if it is always done the same way. For example:

```python
from herschel.ia.pal import *
from herschel.ia.pal.pool.lstore import *
from herschel.ia.pal.pool.hsa import *

urlProducts = ...
urlMetadata = ...
pool1 = LocalStoreFactory.getStore("localPool")  # local target pool
pool2 = HsaReadPool(urlProducts, urlMetadata)    # remote read-only pool
store = ProductStorage([pool1, pool2])
```

Besides, this has the disadvantage of hard-coding the pools and storage creation.

The PoolManager comes to aid this by allowing defining pools in configuration files, and importing them in a session by their name:

```python
from herschel.ia.pal import *

pool1 = PoolManager.getPool("localPool")  # local target pool
pool2 = PoolManager.getPool("hsaPool")    # remote read-only pool
store = ProductStorage([pool1, pool2])
```
The StorageManager finishes it off by allowing defining which pools to register in a storage, so only the storage name is required:

```python
from herschel.ia.pal import *
store = StorageManager.getStorage("myStorage")
```

It may be defined by another configuration file, or just by a property like this:

```python
hcss.ia.pal.store.myStorage = { localPool, hsaPool }
```

**Exporting Products**

Exporting products directly to files, and reading them back, can be done without the need of the somewhat complex structure of PAL pools. Moreover, it is highly desirable to be allowed to import any standard FITS file into an IA session, even if it comes from another source different than Herschel DP.

The Archive interface provides this basic functionality. There are two available implementations, one for saving and loading products in FITS files; the other one by using the Java serialization mechanism:
Processing Framework

In section **Data & Computation Services**, a brief overview of the relations between data structures and the processing is presented. The various computational frameworks that are shown ensure not only common look-and-feel, but also aid developers in writing new implementations of processing entities such as numeric functions, (pipeline) tasks and scriptable pipelines.

The aim is that if a user can understand the syntax and provisions of one type of processing implementation, the learning curve for similar processing implementation diminishes.

This section provides a brief overview of these frameworks:

- **Numeric Framework**
- **Task Framework**
- **Pipeline Framework**

**Numeric Framework**

This section gives a brief overview of the Numeric computational framework. The complete design of the numeric toolbox framework can be found in the *Numeric Library Design Document*.

A set of numeric functions that operate on raw array data and scalar data are required. These functions need to be available in both Java and Jython. A common look-and-feel eases the understanding of the syntax of all numeric functions. In addition, usage of Jython specific convenience constructs simplifies the syntax on the Jython command-line.

**Framework implementation**

The general scheme is that we have a finite set of array implementations as presented in the section **Numeric Structures**. For these implementations, functions need to be developed that are:

- common to all array types or a subset of them
- common to arrays of any rank or a subset of them
- acting as a real function by producing a new result, or acting as a procedure by altering the array itself or both
- sometimes producing an array of the same rank, sometimes of a different rank or even scalar
- sometimes producing a result of the same type, and sometimes of a different type
common to all instruments, or a subset of them

A framework of abstract function framework implementations that follow the visitor pattern can support this kind of flexibility as follows. Consider the relation between the interfaces below:

```java
interface ArrayData {
    ArrayData apply(ArrayToArray f);
}

interface ArrayToArray {
    ArrayData of(Bool1d x);
    ArrayData of(Double1d x);
    ArrayData of(Double5d x);
    ArrayData of(Complex5d x);
}
```

Then a user can execute a particular function of type ArrayToArray (such as SQUARE) as follows:

```java
ArrayToArray SQUARE; // defined elsewhere
ArrayData x; // defined elsewhere
ArrayData y = x.apply(SQUARE);
```

Functions that require additional arguments can provide these through the constructor of that ArrayToArray implementation:

```java
ArrayData y = x.apply(new Reshape(5,3));
```

Note that this setup allows re-use of a pre-defined function, removing the need of unnecessary creation of these functions:

```java
ArrayData[] x,y // defined elsewhere
ArrayToArray f = new Reshape(5,3);
for (int i = 0; ...; i++) {
    y[i] = x[i].apply(f);
}
```

**Jython syntax**

The syntax for Jython users can be eased by introducing Jython Hooks by extending the function interface with the following method:

```java
interface ArrayToArray {
    Object __call__(ArrayData); // Jython hook
}
```

This construct then allows the same functions to be called as:

```java
y = f(x) # e.g. f=Reshape(5,3), f=SQUARE, f=SIN, ...
```

The provision of abstract function implementations for these interfaces within the framework, that automatically throw an Exception for each method that deals with a specific array type and rank, allows an implementer of that abstract function to overwrite only those methods that are required for that function implementation. For example, if a function only applies to double arrays of rank 1, an implementer only has to override one method.
class MyFunction extends AbstractArrayToArray
    ArrayData of(Double1d x) {
        Double1d y = new Double1d(...);
        // do something
        return y;
    }

Toolboxes

Numeric function implementations should be grouped into toolboxes that provide grouped functionality, such as trigonometric functions, fitting functions, random functions, filter functions, transformation functions and so on. In addition, instrument specific toolboxes may be required. The relations between the numeric toolbox framework and the various toolboxes are presented in section Numeric Toolboxes.

Task Framework

This section gives a brief overview of the Task framework. The complete design of the Task framework can be found in the Task Framework Design Document.

Numeric functions are low-level functions that operate on simple array data. These numeric functions are then used to build up higher level data processing functions -called tasks-.

The purpose of the Task framework is to provide facilities within which developers of data analysis software can concentrate on the writing of algorithm implementations rather than the environment they work in. The task framework allows the developed tasks to be seen by the end user with a constant uniform syntax based on function style:

result = task(input=12)

Tasks developed within the framework can be used in different phases of the Herschel mission, as illustrated by the environment used during the phases: Quick Look Analysis (QLA), interactive sessions (IA) and batch processing, like Standard Product Generation (SPG), Trend Analysis (TA), etc.

The use of a common framework allows for a configurable behavior made to adapt tasks to the varying needs of the different environments. For instance, in the IA and QLA environments, graphical user interfaces (GUI) are very useful for querying parameter values from the user, while in batch mode this is a forbidden feature; in a batch session one wants to keep track of the processing steps together with the product they manufacture (history), which is not required for an interactive session, etc.

Framework implementation

The task framework provides the developers with the following features:

- Automatic checking and validation of parameters.
- Allow for standard inspection of the parameters (for example in a Graphical User Interface).
- GUI support for custom interfaces.
- Ensure that general facilities required for processing can be uniformly applied over all tasks:
  - logging,
  - progress monitoring,
  - status recording,
  - history recording.

The task framework provides the end users with the following features:

- A common look-and-feel and standard user syntax: learning to communicate with one task is sufficient to understand the syntax of all tasks.
- Automatic Standard notification of missing/wrong passed parameters.
- Support for user requests of interrupt of execution.
Task Overview

A task is composed of two parts:

Signature
The set of parameters that defines the task interface to the user.

Execution Model
A schema of execution that encapsulates the logic of the task (its algorithm) and provides support for full customizable behaviour.

Task Signature

The signature of a task comprises the set of parameters that define the task user interface (input and results of the task).

The task implementer provides a list of tasks parameter definitions, in the form of key and value type, optionally a default value and a mandatory rule:

```java
public class Task {
    public Signature getSignature();
}
```

```java
public class MyTask extends Task {
    public MyTask() {
        Signature s = getSignature();
        s.addTaskParameter(new TaskParameter("a", new Double(37.2)));
        s.addTaskParameter(new TaskParameter("b", String.class));
    }
}
```

The framework also provides a default set of parameters (system parameter) always available in a task signature that provide support for common features such as status and progress.

Task Execution

A rough sketch of the execution model of the task framework is as follows:

```java
public class Task {

    // to be implemented by task developers in derived classes
    public void execute() {}

    // called by users
    public void perform() {
        preamble();
        execute();
        postamble();
    }

    private void preamble() { /* validate parameters by default */ }
    private void postamble() { /* update the product’s history */ }
}
```

Preamble

The default action of the preamble is to check the validity of the parameter values that the user has provided to the task. This is done by first checking formally the signature and then validating (or not) the actual parameters. The formal verification consists in checking that the parameter has a non empty name, has a default type, and in case it has a default value, this value is of a correct type. The actual validation checker verifies that if the parameter is mandatory, a user value has been provided, checks that the actual value is null only if allowed, and finally checks the compatibility type between the actual value and the parameter type.

Execute

The execute method is empty and is meant to be redefined by the developer of a Task. This is the place where the developer will implement the algorithm using the input parameters (which have been validated) and produce the output parameters.
Postamble

By default the postamble will record the history in the output products if this has been enabled, and will reset the parameters to their default value as requested.

While the model might look elaborated (for an end user) the resulting task interface is extremely simple. If the above example is completed the below is the resulting user interface:

```python
myTask = MyTask()
result = myTask(a = Double(12), b = "option")
```

The execution model is fully configurable i.e. each of the three methods is delegated via a Strategy pattern and can be replaced according to the different needs of the different environments. This is done at the system level and therefore it is completely transparent both to developer and users.

Pipeline Framework

This section gives a brief overview of the Pipeline framework. The complete design of the Pipeline framework can be found in the Pipeline Framework Design Document.

In summary, a pipeline script is a script that applies a set of pipeline specific tasks on input Products controlled by some configuration parameters. The tasks in such a pipeline are chained together by some logic such as if-then-else constructs and they produce Products. In addition to the input Products, tasks may have simple control parameters; these configuration parameters are set within the pipeline, and their values may be extracted from the pipeline configuration.

A pipeline script is that part of the processing that is shared across all applications and environments; in other words, the same script can be used within the operational environment as well as on the astronomers’ computer. This requires that the script must be void of details about its environment.

Note that pipeline scripts should be deterministic; that is, provided that the input products and pipeline parameters are known, the pipeline can access all relevant data and execute the various tasks within the pipeline.

A pipeline Task should be able to return a simple Product output or a Context (which is a Product). If the output is a Context, then we should allow a task to incrementally build up that Context by saving the leafs; e.g. when the total output may be too big to fit into memory.

We could provide the Product Storage API to each task directly, but then one could query and/or alter that ProductStorage e.g. overwriting the results of a previous task. Instead we could provide to the internals of the task framework an interface that accepts a Product and returns a ProductRef. This way we protect the system from various storage weaknesses.

Pipeline Layout

The pipeline layout can be visually expressed as follows:
Pipeline tasks
The blue shapes are the pipeline tasks. They may take one or more products as input and produce one or more products as output.

Products
The gray and yellow cylinders express the input and output products. The yellow products may be considered final output products whereas the gray shapes express potential intermediate products.

Logic
The diamond shape expresses some if-then-else logic that determines the execution path within the pipeline; in this example, it depends on this logic whether Task(D) or Task(E) is executed on the second output of Task(A).

Pipeline Script
Using the example layout above, the pipeline script itself could look like:

```
# short-cuts relevant variables
obs = # an ObservationContext provided, e.g. loaded by User
cfg = # a pipeline Signature provided, e.g. loaded by User
store = ProductSink.getInstance()  # holds the interface for saving data

# running all tasks/logic in the pipeline
outA = taskA(input=obs.telemetry,
```
offset=obs.cal.spire.pixel.offset,
    par1=cfg["a"], par2=cfg["b"])
obs.level1["outA"] = store.save(outA)

# ... Left-hand side of the pipeline
outB = taskB(input=outA[0], par1=cfg["c"], par2=cfg["a"],
              cal=obs.cal, aux=obs.aux)
obs.level1["outB"] = store.save(outB)

outC = taskC(input=outB[0], par1=spg.cfg["c"], par2=spg.cfg["a"],
              cal=spg.cal, aux=spg.aux, saver=spg.saver)
obs.level1["outC"] = store.save(outC)

# ... Right-hand side of the pipeline
if some_condition_is_true:
    outD = taskD(input=outA[1], par1=spg.cfg["x"], par2=spg.cfg["y"],
                 cal=spg.cal, aux=spg.aux, saver=spg.saver)
    obs.level1["outD"] = store.save(outD)
else:
    outE = taskE(input=outA[1], par1=spg.cfg["y"], par2=spg.cfg["z"],
                 cal=spg.cal, aux=spg.aux, saver=spg.saver)
    obs.level1["outE"] = store.save(outE)
pass

# save the updated observation context, preserving a reference
# to the previous version.
store.save(obs)

Note that the observation variable, the configuration variable and store variable are entities that are configured
and provided outside the script itself; they -in essence- are provided by the environment in which this pipeline
script is ran. For example, a user may load a specific observation context into the interactive analysis system,
whereas for systematic processing this may be controlled by a scheduler.

Some additional remarks about the above pipeline scripts:

Pipeline Configuration
The pipeline configuration could be expressed using the Signature facilities as used for Tasks.

ObservationContext as State-Machine
In this example all input products and output products related to this observation are contained within an
ObservationContext which is derived from the Context Product family. In that sense the ObservationContext
is a state-machine: it gets updated and saved after each modification. The ObservationContext is therefore in
a particular state (e.g. complete, incomplete, etcetera). The minimum input state of the ObservationContext is
that it holds the raw input Product as well as the accompanying auxiliary and calibration Products. After the
pipeline has been successfully executed, the ObservationContext holds the resulting level 1 Products as well.

Pipeline Output Sink
The singleton ProductSink.getInstance() in the pipeline script provides access to an entity that allows to
save the output products of each task after the task has been executed.

Output of a Pipeline Task
Multiple output Products can be expressed e.g. by using a Context Product and the Product Reference
mechanism:
# Task producing multiple Products:
class TaskA(JTask):
    def execute(self):
        ...product1, ...productN created
        out = ListContext()
        out.refs.add(ProductRef(product1))  # add child to output context
        out.refs.add(ProductRef(productN))
        setValue("output", out)
        return

Note that the singleton ProductSink.getInstance() in the pipeline script can also be accessed within a pipeline task. This is useful in those cases where a single task may create an output Product which could be too big to fit into memory:

# Task producing multiple Products, with minimal memory footprint:
class TaskA(JTask):
    def execute(self):
        ...product1, ...productN created
        out = ListContext()
        store = ProductSink.getInstance()  # used lazy writing
        out.refs.add(store.save(product1))  # save child and add it to output context
        out.refs.add(store.save(productN))
        setValue("output", out)
        return
Product Generation Framework

The Product Generation Framework provides the functionality to:

1. gather relevant data from object-oriented (ObjectStore) as well as product-oriented databases (PAL),
2. process that data from its rawest form down to Products that the pipeline framework can understand,
3. execute a pipeline specific to that observation using the same modules as will be provided to the astronomer with the use of default control parameters,
4. produce quality data and to inspect the quality of that data and
5. ingest all results back into the database.

This framework works on an observation by observation basis; that is, it expects input data relevant to a single observation and generates Products specific to that observation. Applications can use this framework to configure and execute a complete data reduction cycle on a single observation.

The Product Generation Framework consists of three distinct phases:

Pre-processing

In the first part of the framework execution, data may be available in the ObjectStore (aka the object-oriented-database), such as telemetry and/or dataframes or within the PAL, such as calibration data, auxiliary data. Therefore one should assume in this phase the availability of a connection to the OO-database as well as to the PAL, which in turn may be configured to point to the HSA.

Observer’s Pipeline

The next step is the execution of the so-called Observer’s Pipeline, which must be done in an object-oriented-database free environment, such that an observer can run this processing (or parts of it in the form of its modules) in a local environment (aka a laptop). Therefore the only connection will be the PAL.

As such and prior to running the Observer’s Pipeline, any ObjectStore specific data must be transformed into Products that can be accessed via the PAL. This is the so-called harvesting of Products and generation of Level0 Products collected into an Observation Context.

Post-processing

In the post-processing phase a quality inspection and ingestion into the Herschel Archive is achieved. This is a responsibility of the Herschel Operational Team and in this phase one can again assume the availability of the PAL, which will point to the HSA in the environment of the Systematic Product Generation.

The following diagram expresses the data flow of an Observation Context:
Pre-Processing

The pre-processing phase distinguishes three major components:

- Finding the relevant Auxiliary Product Tree specific to this observation
- Finding the relevant Calibration Product Tree specific to this observation
- Generation of Level0 Products from Telemetry and/or Dataframes.

Prior to the execution of these components, one should assume the availability of the following identifiers within the Observation Context:

- `obsid` # observation id (long)
- `day` # observation day (int)
- `startTime` # start time (FineTime)
- `endTime` # end time (FineTime)
- `instrument` # instrument name (String)
- `modelName` # instrument configuration (String)

If such code would be written in a Observation Locator plugin, then the Product Generation framework does not need to know about the actual mechanism to generate this data. In fact it can be envisaged that at least two types of situations may exist:

- An ObservationContext needs to be created from scratch (SPG) or
- An ObservationContext already exists in the system, but needs to be reprocessed automatically (BPG) or on-demand (OPG).

These parameters can be deduced from the Core Class Model by navigating the object oriented database as follows:

```java
// Java pseudo code!
ObjectStore store = StoreFactory.create().createStore(HOD);
CoreFactory coreFactory = CoreFactoryManager.getInstance();
ObservationExtent extent = coreFactory.getObservationExtent();
Observation observation = extent.getObservation(store, obsid);

ObservationContext oc = new ObservationContext();
oc.setId(obsid);
oc.setStartTime(observation.getStartTime());
oc.setEndTime(observation.getEndTime());
```
InstrumentModel model = observation.getInstrumentModel();
oc.setInstrument(model.getInstrumentName());
oc.setModelName(model.getModelName());

From this information the above mentioned components can find the relevant information and generate level0 products.

These components could be written as plugins; at start-up of SPG the system reads a configuration of available plugins which are then picked up and executed at run-time. This has several advantages:

- It decouples SPG framework code from algorithm code. The logic how to find or how to generate data is pushed out of the SPG framework, keeping the core clean
- It avoids impossible circular dependencies. Major parts of the code must be written by the Instrument teams and moreover are instrument specific (e.g. generation of level0). This code will most probably reside within the instrument specific code hierarchies which are invisible to the core system at build time
- It allows to implement different plugins which can be selected at run-time. Allowing the plugins to be configurable, one can write different plugins (e.g. formal ones, one for debugging, etc...), making the system quite flexible.

**Auxiliary Product Finder Plugin**

The algorithm to identify the relevant Auxiliary data requires at a minimum two identifiers: observation id and observation day. These identifiers should be available in the first state of an Observation Context Product. With this information a plugin for e.g. the SPG can be written that takes this state of the Observation Context and with a given PAL setup, the appropriate auxiliary data can be found and added to the Observation Context. The framework records the new state of the Observation Context.

```java
// Pseudo code!
oc: ObservationContext (defined)
pal: ProductStorage within the PAL (defined)
findAuxiliaryContext(oc, pal);
pal.save(oc);
```

**Calibration Product Finder Plugin**

The algorithm to identify the relevant Calibration data requires at a minimum the identifiers: instrument, modelName, startTime and endTime (for applicability of the calibration data). These identifiers are available in the first state of an Observation Context Product. With this information a similar plugin as described for the auxiliary data finder can be written.

```java
// Pseudo code!
oc: ObservationContext (defined)
pal: ProductStorage within the PAL (defined)
findCalibrationContext(oc, pal);
pal.save(oc);
```

Ideally all instruments would have the same calibration hierarchy and high level interface: in such case only one algorithm could be written. However this may not be the case (especially in early phases of development).

**Level0 Generator Plugin**

The algorithm to generate instrument specific level0 data, probably requires all elements that are described above plus access to the CCM: Observation entry point in order to convert Dataframes and/or Telemetry into Level0 Products.

```java
// Pseudo code!
oc: ObservationContext (defined)
pal: ProductStorage within the PAL (defined)
store: ObjectStore (defined)
generateLevel0(oc, pal, store);
pal.save(oc);
```
Ideally, the only thing that would be passed to this plugin would be some form of iterator over all Telemetry and/or Dataframes. Such way, the plugin could be written void of knowledge of the OO-dbase:

// Stripped down and independent interface:
interface ObservationBlock {
    long getBlockId();
    Iterable<Dataframe> getDataframes();
    Iterable<TmSourcePackets> getTelemetry();
}

// Pushed somewhere else:
Iterable<ObservationBlock> rawData = getObservationBlocks();

// Instrument specific code only accesses:
generateLevel0(oc, pal, rawData);

Observer’s Pipeline

This part of the framework is executing code that can be run within Operations as well as on the Observer’s laptop. The data has been prepared in the pre-processing phase and made available through the PAL in such way that one can run this pipeline in any environment.

Details of the pipeline framework are discussed here.

Note that this framework has to deliver to the observer’s pipeline the following entities:

Active Observation Context
A reference to the active observation context, which is created and/or loaded in the pre-processing phase. This is provided in a variable called "obs".

Active Pipeline Signature
This is a pointer to configuration parameters of this active pipeline.

Active Output Sink
With this we mean an entity where lazily written data (e.g. children from context products) can be stored. Note that the user can always load and save Product data through the PAL at any time, but this may be prohibited by the SPG within operations, in order to avoid that the observer’s pipeline is not working in a deterministic fashion.

The latter can be provided in statically in the form of e.g. ProductSink.getInstance(). Such way it does not require modifications to the task framework that is required to write SPG modules (aka tasks); a SPG module can simple call:

```
public MySpgTask extends Task {
    public void execute() throws InterruptedException {
        // the output product, lazily constructed
        ListContext output = new ListContext();
        ProductSink store = ProductSink.getInstance();
        ProductRef child = null;

        child = store.save(createChild1());
        output.getRefs().add(child);

        child = store.save(createChild2());
        output.getRefs().add(child);

        setValue("output", output);
    }

    private Product createChild1();
    private Product createChild2();
}
Post-Processing

Uses similar plugin facilities as in pre-processing to allow the harvesting of quality data from the various levelX products in order to collect and summarize quality information into the quality product.

Plugin Design

This section briefly walks through the Product Generation plugin design:

- What is required
- Plugin Framework Interfaces
- A Plugin Implementation
- Plugin Manager Implementation
- Plugin Usage within SPG

What is required

1. Pre-processing as well as post-processing require access to both the ProductStorage (part of the Product Access Layer) as well as the ObjectStore (part of the object oriented database system), one being an astronomers oriented storage system and the other using a specialized object-oriented database. It is therefore required that the plugin mechanism mentioned previously should provide facilities to access both of them.
2. All plugins of all instruments required to allow the PG system to access calibration data and generate level0 products etcetera should be configurable via properties.
3. As all plugins need to be registered, the construction phase of these plugins should be light weight, that is short construction time and minimal foot print.
4. Identification of plugins should follow a strict convention such that they can be automatically picked up by an application of this framework when access to a specific plugin is required.
5. Plugins should be able to automatically register themselves.

The above requirements hint to a design embracing a plugin manager, a plugin interface and a way to pass the observation context and the required access to both the ObjectStore and the PAL.

Plugin Framework Interfaces

The plugin design could be expressed by the following interfaces:

```java
public interface PgPlugin {
    void populate(ObservationContext observation, ProcessEnvironment environment);
}

public class PgPluginManager {
    public static PgPluginManager getInstance();
    public PgPlugin getPgPlugin(String urn);
    public void register(String urn, PgPlugin plugin);
}

public interface ProcessEnvironment {
    public ProductStorage getProductStorage();
    public ObjectStore getObjectStore();
}
```

A Plugin Implementation

A plugging can automatically register itself if its implementation follows a static initialization strategy:

```java
package herschel.instrument.spg;
public class SamplePlugin implements PgPlugin {
    public static final String URN;  // contract on naming convention!
```
// contract on static initialisation!
static {
    PgPluginManager.getInstance().register(URN, new SamplePlugin());
}

private SamplePlugin() {
}

public void populate(ObservationContext observation, ProcessEnvironment environment) {
    // Implementation code
}

Plugin Manager Implementation

The Plugin Manager retrieves all required plugin implementations from a property. A real life property listing could be:

hcss.ia.pg.plugins = {
    herschel.ia.spg.AuxPlugin,               # finds necessary auxiliary data
    herschel.pacs.spg.plugin.CalPlugin,      # finds necessary pacs calibration data
    herschel.pacs.spg.plugin.Level0Plugin,   # generates pacs Level0 product
    herschel.pacs.spg.plugin.QualityPlugin,  # harvests pacs quality data
    herschel.hifi.spg.plugin.CalPlugin,      # finds necessary hifi calibration data
    herschel.hifi.spg.plugin.Level0Plugin,   # generates hifi Level0 product
    herschel.hifi.spg.plugin.QualityPlugin,  # harvests hifi quality data
    herschel.spire.spg.plugin.CalPlugin,     # finds necessary spire calibration data
    herschel.spire.spg.plugin.Level0Plugin,  # generates spire Level0 product
    herschel.spire.spg.plugin.QualityPlugin  # harvests spire quality data
}

The property is accessed once, and all the plugins are read as follows:

// Actually a singleton, but only plugin mechanism is shown here:
public final class PgPluginManager {

    public static final String PLUGINS = "hcss.ia.pg.plugins";
    private boolean _registryRead;
    private Map<String, PgPlugin> _registry;

    public PgPlugin getPgPlugin(String urn) {
        readProperties();
        return _registry.get(urn);
    }

    public void register(String urn, PgPlugin plugin) {
        _registry.put(urn, plugin);
    }

    private void readProperties() {
        if (!_registryRead) {
            List<String> classes = Configuration.getList(PLUGINS);
            for (String clazz : classes) {
                Class.forName(clazz);  // calls the static initializers
            }
            _registryRead = true;  // read once!
        }
    }
}
Plugin Usage within Framework

Within the PG framework implementation, the various plugins could then be picked up

```java
public class ProductGenerator {
    :
    :
    // observation already got populated with things like instrument...
    private void preProcess(ObservationContext observation) {
        // populate with spacecraft data:
        PgPlugin aux = getManager().getPgPlugin("urn:aux:sc");
        aux.populate(observation, getEnvironment());

        // the instrument name we use for accessing the instrument specific plugins:
        String instr = observation.getInstrument().toLowerCase();

        // populate with instrument specific cal:
        PgPlugin cal = getManager().getPgPlugin("urn:cal:" + instr);
        cal.populate(observation, getEnvironment());

        // fetch instrument specific cal:
        PgPlugin level0 = getManager().getPgPlugin("urn:level0:" + instr);
        level0.populate(observation, getEnvironment());
    }

    // internal convenience method:
    private PgPluginManager getManager() {
        return PgPluginManager.getInstance();
    }
}
```

Observation Locator

The observation locator plugin works slightly different from the above plugins, in the sense that the PG framework can only register one at a time and the plugin may also be responsible for creation of the initial ObservationContext if an existing one can not be found.

```java
/**
 * Finds or creates an ObservationContext for a given obsid and instrument,
 * and within the given environment.
 * Only a single ObservationLocator can be used within SPG.
 */
public interface ObservationLocator {
    public ObservationContext find(long obsid, Instrument instrument, ProcessEnvironment env);
}
```
**Visualization Framework**

In section Data & Visualization Services, a brief overview of the relations between data structures and the visualization mechanism is presented. The aim of the visualization framework is to provide a unified mechanism for inspecting and possibly editing of products, datasets and array data, and this irrespective of their structure and content. The different components in the visualization framework should seamlessly work together to provide an efficient inspection platform for data content and structure.

**General Concepts**

Visualization of any data structure and its content will implement the Model/View/Control design pattern. This pattern decouples the data model from its representation and provides a separate user interface for controlling its representation. Data structures will need to provide a communication channel to their Views in order to ensure an up-to-date representation of the current state of the model. The pattern that will be used here is the Observer design pattern which in Java is provided by a Listener interface.

Data structure in Herschel DP are very modular and build up recursively from different smaller data structures. We want to minimize the maintenance on Viewers and maximize their reusability. Viewers will therefore be built up using the Composite design pattern. With this pattern, complex Viewers can be built from other more simpler and basic Viewers, but have the same interface when it comes to integration in the visualization framework and applications.

Ideally the Views should allow different Controllers to interact with the user. This can be accomplished with a strategy pattern by which different Controllers can be changed either statically or dynamically. This will increase the user friendliness and common look and feel of the visualization system for Herschel DP.

**Visualization of Data Structures**

Different levels of data aggregation are associated to different means of visualization. The following paragraphs summarize the ways for viewing the available data structures and their storages.

**ArrayData Visualization**

Array data exists in different types and dimensions going from a one-dimensional string array to a five-dimensional double array (see Numeric Data Structures). All these types and dimensions each have their specific ways to visualize their content and structure. Raw views of array data can display a one- or two-dimensional slice of a multi-dimensional cube in a tabular view, but even so can two-dimensional arrays be displayed as a color mapped image or as a 3D wireframe surface plot.

The framework will provide basic 2D plotting tools for plotting time series, XY-plots, histograms, etc. For two-dimensional array data an image display tool will be provided which can also be used to display two-dimensional slices of multi-dimensional array data.

**Product & Dataset Visualization**

Array data are optimized for numeric performance and do not carry too much baggage like meta data and structural information. That is the role of datasets and products. This additional information and the way datasets and products are structured need specific visualization tools. There is a limited set of basic datasets and products defined in the DP framework. More complex and specialized datasets and products can and will be constructed from these basic building blocks. The framework provides means to visualize all these data structures in a consistent way, using above mentioned concepts.

The visualization framework also provides a view on the data structures that are defined in an interactive session, allowing the user to choose, inspect and possibly manipulate these structures.

The tool that is envisaged to provide this functionality is the Inspector. It will implement a plugin architecture for views on data structures, visualizing which data structures are defined in your working session and how they are structured and possibly related. The Inspector also provides plugins for exploring the data content and a mechanism to allow Views and Explorers to communicate and exchange information about data structures.
Default Views will be provided to list available datasets and products and for each basic dataset or product type a default explorer plugin is provided by the framework. When new, more complex data structures are constructed, associated Explorers can be developed to visualize these data structures. The plugin mechanism makes such explorers known by default to the active Viewers.

**Product Storage Visualization**

The Storage Framework describes the functionality provided to store and retrieve data products from a local store or a remote database. On top of this framework resides a graphical user interface which serves as a visual user front-end to communicate with the Product Access Layer (PAL) without the need to know the complete API for querying and loading data products. The Product Browser works from a given storage.

The Product browser provides a querying area where the user can construct complex queries with just a few simple mouse clicks. Queries can be on product attributes or meta data, but also full data mining queries can be entered in a special text field. The query can be restricted on a specific product type and can be refined from the result of the previous query.

A second area in the GUI shows the result of the query as a list of matched products. This list can be customized to show only the columns of interest. Products can be selected and will show up in the JIDE basket which is the communication channel to the users interactive session. Each product in the query result list can also be visualized re-using a dedicated explorer of the same type as used in the Inspector described above.

**GUI Application Elements**

A GUI application in Herschel DP may be based on a GUI framework that provides means for creating views, as well as means for arranging them in different ways for different purposes, in what is called perspectives. Moreover, for facilitating user interaction, the views should provide toolbars that shall be configurable in an easy way by developers.

A special kind of view is the editor area, which can hold many sub-views for viewing/editing script files, tasks, images, etc.

The following picture shows these concepts graphically:
Views

Views provide the means to look at the resources -or parts of them- in different ways. One can think of a console, a help window, an overview of all variables in the session and so forth. Some views are extendible, such as the Outline View. All views are contained within a view part, which provides the drawing box, an icon and a label.

Creating a view implies implementing the `Viewable` interface by extending a `JComponent`, and registering it in an initialization file with a unique name, so the application is aware of its existence.

The actual creation and population of the view is done in an automatic way for all the registered views. The final rendering of a view in the GUI is made internally, and may be delegated to a third party library, which can be then changed without altering the public contract:
Editor Area

The editor area is the location for visualizing the contents of user resources. Resources may be an image, a spectrum, dataset or a product. Another type of resource can be a user script, or a task execution. The editor can show e.g. an ImageDataset as an image or as a browsable dataset.

Each sub-view within the editor is called an editor part, which contains an editor component that is responsible for displaying and managing the corresponding selected resource:
Perspectives

A perspective is a collection of graphical windows organized in a way to focus the user on doing a specific job within the whole suite of jobs that a user can and will do within the system. It may consist of one or more views and optionally the editor area. Examples of standard perspectives are a full-fledged work-bench, the PAL query perspective, and the classic JIDE perspective.

A perspective must implement the `Perspective` interface, as well as be registered in an initialization file with a unique name, so the application is aware of its existence.

The widgets within a perspective are organized in tabbed panes and split panes. The building blocks of a perspective are called site parts, which can be then any of the following: view part, editor area, split part or tab part. The two latter are just means for dividing parts into the available space, either horizontally or vertically, and allowing different views appear in the same area, by being selectable with tabs (the editor parts within the editor area are an example of a tab part).

This mechanism may be aided by a helper tool like the following:

```java
public interface ViewPartManager {
    EditorArea getEditor();
    ViewPart getViewByTitle(String title);
    ViewPart getViewById(String id);
}

public interface SitePartBuilder {
    ViewPartManager getViewManager(); // for getting views and the editor area
    TabbedPart buildTabs(SitePart... parts);
    SplitPart buildSplit(float split, boolean horizontal, SitePart lhs, SitePart rhs);
}
```

Menus and Toolbars

Interactively acting on the GUI may be aided with the use of menus and toolbars:

- The application contains a main menu, each view can have its own drop down menu, and any visual component may have associated a pop up menu.
- A toolbar is a set of buttons normally labeled with an icon and void of text. Similarly, there would be a main toolbar, and each view may contain its own toolbar.
The main menu and tool bar are expected to give access to global functionality as well as to the editor area. On the other side, the view’s menu and tool bar provide access to functionality which is specific to that view.

There are pre-defined actions, like cut, copy, paste, save, etc. that are common to all applications. These actions are already defined, but the implementer of a view can redefine (retarget) its behaviour to its specific purposes. This behaviour is made available when the view becomes the focused one.

The menu manager helps with handling actions along menus and toolbars:

```java
public interface ActionBars {
    // Retarget existing actions
    void retarget(Retarget type, SiteAction action);
    void retarget(Retarget type, String actionId);

    // Insert new actions
    void insert(Insert insertion, Retarget retargetable);
    void insert(Insert insertion, String actionId);
    void insert(Insert insertion, SiteAction action);

    // Remove existing actions
    void remove(Retarget retargetable);
    void remove(String actionId);
    void remove(SiteAction action);
}
```

```java
public interface MenuManager {
    // Access to visual components
    JMenuBar getMainMenu();
    JToolBar getMainToolBar();
    JToolBar getToolBar(String view);

    // Object for inserting/retargeting actions
    ActionBars getActionBars(String id);

    // Any component interested in displaying a pop-up menu
    // for mouse right-clicks needs to register this mouse listener
    MouseListener getMouseListener();
}
```

**Events**

Views and editor components can communicate each other by means of events. Triggering and listening to events is done through an event handler provided by the view part:

```java
public interface ViewPart() {
    SiteEventHandler getEventHandler();
}
```

```java
public interface SiteEventHandler {
    // Sends an event to all interested parties
    void trigger(SiteEvent event);

    // Registers a listener to events of a specified type
    void addEventListener(Class<? extends SiteEvent> type, SiteEventListener listener);

    // Removes a listener from the specified type
    void removeEventListener(Class<? extends SiteEvent> type, SiteEventListener listener);

    // Removes a listener from all the registered types
    void removeEventListener(SiteEventListener listener);
}
```
public interface SiteEventListener {
    void selectionChanged(SiteEvent event);
}
Interpreter Framework

Services available within the Herschel Data Processing as:

- Processing
- Product Generation
- Storage
- Visualisation

which are developed and natively available with the grammar and syntax of the Java language, are also made available to end user via a dedicated Interpreter.

Interpreter advantages

The availability of an Interpreter allows for the following advantages:

Dynamic types
New variables can be created without predetermining the variable type. The interpreter performs the required checks and type conversions automatically.

Compilation on the fly
The Interpreter performs the script compilation and execution within the same process.
Usually, the interpreter also parse and compile scripts into intermediate code when they are first executed.

Simplified syntax
Due to the above features there’s no need for a strict specification of the types of variables, moreover it is possible to simplify many other language constructs.

Interactivity
An interactive interpreter allow for “trial and error” data reduction. While errors generated during the execution of a program usually don’t allow for a recovery, in an interactive interpreter the failure of the execution of a command can be recovered and it is actually integral part of the data reduction which can continue from the point where the error occured.

Jython Interpreter

Many interpreters provide an interface to the Java language so that there is no need to develop an ’ad hoc’ interpreter.

Jython has been chosen from the available interpreters for the following reasons:

Python syntax
Jython is complaint with a language which is already a popular language in the scientific enviroment.

Seamless access to Java
End user can use the powerful Java class library to extend the abilities of Jython. The synergy between the Java platform and Jython language produces an environment in which developers and end users can collaborate to create more useful, dynamic applications. As for instance the numeric framework is designed such as many of its features appear as original Jython features.

Implemented in Java
Jython is written in Java, from the Java point of view it is just another library. That allows to use the interpreter features into another parts of the framework, like in the Storage for implementing its query language.

In the Data Processing environment the Jython is pre-configured to load and recognize all the Data Processing libraries.

Command Line Interface

The services of the Jython interpreter are exposed to the end user via a dedicated user interface. Such interface is the entry point for many of the applications of Data processing such as CIA, OIA and QLA.

The complete Command Line Interface is composed of different components:

Console
The main component that interacts with the Jython Interpreter. It offers command line execution, command history storing and recall, buffered outputs (in space and time) for messages coming from the interpreter.
Command Editor
   A component for visually displaying, save history and execute scripts.
Script Debugger
   A component for loading, visualizing, editing, run (also in line by line mode) scripts.
Scripts Pane
   A component for handling multiple scripts (each in its own ScriptDebugger).
Main Component
   A component that assembles all the above components in a congruent view.

The Command Line is also an hub or entry point for most of utilities of the Data Processing Framework that presents a user interface such as some of the visualization utilities, a documentation viewer and a logger.
Documentation Framework

The documentation framework provides tools to help in the construction of the different manuals that describe the complete data processing software for users and developers. Documentation is written in DocBook XML which is a XML language designed for writing technical documentation. Reference manuals are generated from documentation in the source code which can be javadoc for the Java language or special jtags used in Jython code. The results of processing all these inputs can be presented in different output formats; PDF for printing and HTML for on-line viewing.

The figure below gives an overview of the most important documents within the data processing environment. The contents and purpose of each document is explained further.

End-user Documentation

Documentation for end-users, i.e. Astronomers, Calibration Scientists and Instrument Engineers who merely wish to use the DP software, and not necessarily write code or provide scripts for DP purposes.

What's new

The What's New document summarizes the new features available in a given release of the DP software. The purpose of the document is to provide users a quick overview of what is new between DP releases.
FAQ

The FAQ provides answers to the most common questions expected from users, and is intended to be the first place users should look before contacting the Support Helpdesk.

Users Manual

The Users Manual provides all the necessary documentation to enable an end-user to get into grips with using the system. This covers information from basics to advanced aspects of a particular topic. The users manual provides information on the installation of the software, the usage of the interactive environment (JIDE), basic Jython programming constructs and scripting, an explanation of the generic data structures, numeric functionality and an explanation of the visualization tools. This is in contrast to the Users Reference Manual (URM) which is intended to list the major functions available to the end-user, along with summary information for each function, task or data structure.

Users Reference Manual

The Users Reference Manual provides end-user level information on all the functions and task in the data processing environment. The manual is organized to list all functions or tasks alphabetically and by category.

In addition, the URM contains a chapter on numerics functionality, with an in-depth description of numeric arrays and the numeric toolboxes, and a reference chapter on the product access layer.

Developer Documentation

Developer Manual

The Developer Manual (DM) provides a high level description of how to develop software for data processing. This includes software access, installation, and usage of the version control system (cvs). It provides an overview of the development cycle adopted for DP development, how to contribute to the numerics package and write tasks for both standard processing and interactive usage. The DM contains information on using the documentation framework to process the different manuals, how to set up a test environment to run unit tests. There are chapters with specific guidelines for pipeline development, writing tasks, using class loaders and the logging mechanism.

Developer Reference Manual

The Developer Reference Manual is what is called the Javadoc. This documentation is written by developers and provided as part of the source code. It explains the interfaces and the functionality of all the Java classes in DP.

Package Developer Guide and Package Design Document

Each Java package contains in-depth documentation for developers and maintainers of these packages. This documentation explains how the package is designed, what are the dependencies within the package itself and with other packages in the system. These documents contain all the information a developer needs in order to take over the active development or the maintenance of a Java package.
Observation Data Hierarchy

This chapter describes the Observational Data Hierarchy.

Observation Context

An Observation Context is a container of Products applicable to a specific observation. It provides associations to products which are specific to a single observation (e.g. Telemetry Product, and reduced data products) as well as associations to Products that are applicable to multiple observations (such as the calibration products).

As User API

Command-line as seen by the user strive for intuitiveness. One could think of a User command-line API as the following:

```
Observation Context:
obs # loaded, or created elsewhere
obs.obsid     # observation id
obs.odNumber  # operational day number
obs.mode      # observation mode
obs.instrument
obs.modelName
obs.startTime
obs.endTime

obs.telemetry... # product (probably a context product)
obs.auxiliary... # a context product
    .pointing
    .outOfLimits
    .timeCorrelation
    .missingTelemetry
...
obs.calibration... # a context product
    .offset...
    .photometer...
    .spectrometer.oeZpd
    .pixmask...

obs.level["level0"]...
obs.level["level1"]...
obs.level["level2"]...
    .quality...
```

Not only will the observation context be used in SPG, but also in other environments such as the OIA. The design should also facilitate ease-of-use; that is, a user should be able to change aspects, such as associating different calibration data in a simple fashion:

```
obs.calibration.spire.offset = myOffsets
```

A State Machine

An ObservationContext may have a state of completeness, which is defined by the processing of the data for that Observation.

- It is created
- Auxiliary data attached
- Calibration data attached
- Level0 data generated
- Level1 data generated
One thing and another can be depicted as follows:

ObservationContext as a state-machine: Overview

Thus the observation context changes its nature along the way of processing. Ideally, certain aspects of the Observation Context should be invalidated when associations are modified; for example if the Calibration data is changed, the available LevelX data should be invalidated as it was generated using different Calibration data. However it should be possible to get hold of the previously generated LevelX data.

The Observation Context has a metadata key 'obsState' to store/retrieve the current observation state. The value is set by the user (usually, after a processing is finished). The Observation Context performs some verifications in order to be consistent with the data that it have already stored.

See the 'Developer Guide' at the Observation Context documentation.

ObservationContext Design

The Observation Context stores references to other Contexts (Telemetry, Calibration... etc.) instead of storing the objects inside itself.

So, the Observation Context extends from MapContext (which can store references using key/value pairs where the value is not an object but a reference to an object) and uses get/set methods to retrieve and store the other contexts.

When an object is passed in a 'set' method, a reference to this object is created an stored (the reference) into the Observation Context. When a 'get' method is called, the referenced object is returned (not the reference).

This is an extract of the ObservationContext class:

(The code is not accurate, it is just to show the way to work with it. You should see the java class ObservationContext to have a complete view of the code.)

```java
class ObservationContext extends MapContext {
    long getObsid() { ... }

    Context getTelemetry() {
        return (Context)getRefs().get("telemetry").getProduct();
    }

    Context getAuxiliary() {
        return (Context)getRefs().get("auxiliary").getProduct();
    }

    void setTelemetry(Context telemetry) {
        getRefs().set("telemetry", new ProductRef(telemetry));
    }

    void setAuxiliary(Context auxiliary) {
```
getRefs().set("auxiliary", new ProductRef(auxiliary));

The input/output arguments used for contexts are Context objects. The user must extend from Context (or a subclass of Context like MapContext, ListContext... etc.). See ‘Working with the Observation Context’ at Observation Context documentation.

Contexts have several useful capabilities like cloning and versions, related to a Product Storage.

The following picture is a view of the ObservationContext class diagram and the relationships with other contexts:

The label contains in the previous image means has a reference to, as it is explained before (the Observation Context has references to other Products or Contexts). In the picture, the Observation Context and the other contexts extends from Context Product. This is an example, as the Context class is the base class that can store references. In fact, the Observation Context extends from MapContext (a subclass of Context) in order to have associations between references and keys.

**Telemetry Context**

The Telemetry Context is another Context which implements an iterator. In that way, the data can be accessed on demand and it is not necessary to have in memory all the information. The following code shows the procedure from the point of view of an user:

```python
# jython
obs #loaded, or created elsewhere
for packet in obs.telemetry.iterator():
    print packet.apid
    pass
```

In the following picture, the Telemetry Context (TmSourceContext) is the object to be stored into the Observation Context (actually, a reference to it). As the Telemetry Context implements an iterator, the user can go through each Telemetry Product (TmSourceProduct) which is related to the data stored by the CCM.
Auxiliary Context

The Auxiliary Context, like the Observation Context, stores references to other objects, which can be accessed through its corresponding keys (so, the Auxiliary Context extends from MapContext). In this case, the Auxiliary Context has references to Product sub-classes like PointingProduct, OOLProduct, TimeCorrProduct, etc.

The way to access to the Auxiliary Products could be:

```plaintext
# User interface
obs #loaded, or created elsewhere
timeCorrProduct = obs.auxiliary.timeCorrProduct

// Java implementation
```
class AuxiliaryContext extends MapContext{
    public PointingProduct getPointingProduct(){
        return (PointingProduct)getRefs().get("Pointing").getProduct();
    }
    public OolProduct getOolProduct(){
        return (OolProduct)getRefs().get("OOL").getProduct();
    }
    public TimeCorrProduct getTimeCorrProduct(){
        return (TimeCorrProduct)getRefs().get("TimeCorr").getProduct();
    }
    ...
}

The Auxiliary Context return an specific Product sub-class for each referenced object.

See the Auxiliary Context documentation.

**Calibration Context**

The Calibration Context is instrument specific. It means that there is no `CalibrationContext` common class, but each ICC is responsible of creating its own implementation of `Context` for this.

The Calibration Context is built and put in the ObservationContext in the calibration plugin within the SPG.

**LevelX Context**

The products generated at every level of data reduction are instrument specific.

Therefore, they are built in the corresponding plugin of the pre-processing phase (level0) or pipeline in the processing phase (levelX) within the SPG.

**Quality Context**

The Quality Context is, like the ObservationContext, a container of references to other Context or Products. In this case it is used to store any information relative to the quality of the Product generated during the processing of an observation. Internally it is structured following the different levels in what these products are classified but the exact information at these level is left open to fit any possible requirement of the different instruments. Due this, the unique restriction about the classes stored under each one of theses levels is that they must inherit from the Product class, allowing the developer to choose between the implementation of a custom class containing the specific quality data or adopt the same mechanism used already into the ObservationContext and based on the Context subclass, more flexible when we are working with bit amounts of data.

The Quality Context includes also some specific fields as can be the one used to keep track of the comments that the SPA and other reviewers include into the quality report during the evaluation of the observation or the current state of the review process itself.

See the ‘Quality Context’ explanations at the Observation Context documentation.
Applications

This chapter describes the major applications within the Herschel Data Processing software.

OIA
   Observers Interactive Analysis
CIA
   Calibration Interactive Analysis (and Trend Analysis)
SPG
   Systematic Product generation
OPG
   On-demand Product Generation
BPG
   Bulk Product Generation
QLA
   Quick Look Analysis (and Real Time Analysis)
Observer Interactive Analysis

Observer Interactive Analysis (OIA) is a collection of tools supporting the general astronomer to reduce scientific data. Astronomers will use OIA to analyse data products to obtain reduced scientific data ready for publication.
Calibration Interactive Analysis

To be elaborated

The Calibration Interactive Analysis (and its accompanying fellow: Trend Analysis) system is used to provide insight into the instrument and to generate products which will be subsequently used by the Product Generation mechanisms, such as -but not only- SPG.

Trend Analysis uses housekeeping and science data to analyse trends. Trend Analysis is done on calibrator outputs, detector characteristics, etc. to study trends in the instrument behaviour. There is some overlap between TA and CIA in that TA will be required for a full understanding of the instrument calibration.

Environment

To do this kind of analysis, a Calibration Scientist requires access to:

- Herschel Operational Database (using the HCSS Object Store interface),
- Products in the Herschel Archive
- Instrument Specific Product Archives
- Scientist’s local Product Store
- Any other data required for performing CIA/TA and imported into CIA

Such a setup would require a Local PAL setup that may look like (exemplary only):

```python
pal.register(local-store) # read/write
pal.register(icc-archive) # read        cached for off-line usage
pal.register(ha-archive)  # read        cached for off-line usage
```

Dataflow

![Dataflow Diagram]

Try to extract here that the input data may stem from various sources (Operational PAL, CCM, model, imported data) which in turn needs to run calibration/TA specific algorithms to produce new/updated Calibration Products (CPs).

The CPs are stored locally, and may need competition with previous CPs.

Then an announcement is ‘triggered’ and a procedure takes care of ingestion into the Operational Archive.
Systematic Product Generation

The Systematic Product Generation application (SPG) provides the functionality to systematically process observations once all input data have become available. It provides functionality to:

- Queue the processing of Observations
- Prioritize the execution of processing one observation over another
- Be able to distribute the processing over a grid of computer systems in operational environment
- Monitor the progress
- Catch errors in the processing of an Observation

The execution of a single observation is relying on the Product Generation Framework configured suitable for Systematic Product generation.

Storage Configuration

The SPG will be configured such that all observation executions have access to:

- Relevant Telemetry and Dataframes through ObjectStore and CCM API
- Relevant Calibration and Auxiliary data through the PAL API.

Plugin Configuration

It will also be configured to have access to all pre-processing plugins as well as post-processing plugins suitable for Systematic Product generation, giving access to:

- HSC provided plugin for creation of new or finding and loading of existing ObservationContext.
- HSC provided plugin for fetching Auxiliary Products relevant to a specific Observation
- Instrument Team provided plugin for fetching Calibration Products relevant to a specific Observation.
- Instrument Team provided plugin for generation of Level0 Products for that Observation.
- Instrument Team provided plugin for automated harvesting of quality data and generation of Quality Products specific to the LevelX Products relevant to that Observation.

Pipeline Configuration

The Observer's Pipeline (being the core of the data reduction) should be executed for given observation, which means that at a minimum the SPG requires:

- A pipeline script for processing data for a single observation
- An accompanying default configuration to execute that script

Note that the instruments run in a particular mode, so it is expected to exist one pipeline per instrument mode. Besides, an observation may run two instruments at the same time (e.g. PACS/SPIRE parallel mode). In this particular case, an observation has associated two ObservationContexts, one per instrument, and a pipeline is thus needed for each instrument. The SPG will handle these parallel pipelines in two different processes.
On-demand Product Generation

The On-demand Product Generation Application provides the functionality to allow the end users to process observations into the computing structure available by the Herschel Science Center and get access to the result of the computation.

Specifically it provides access to the following functionalities:

- User Validation and Authentication
- Inputs Selection
  - Calibration Selection
  - Pipeline Selection
  - Parameters Selection
- Output format selections
- Computation of request
- Notification of completion
- Visualization results

The On-demand Product Generation Application is composed of two main components:

- User Front End
- Backend Engine

This architecture decouples the application according to the Client-Server paradigm.

The two component are described in the below sections.

User Front End

The User Front End component is responsible for the implementation of the functionalities dealing with the user interaction, functionalities required before and just after the completion of a request, effectively all above mentioned points except the "Computation of request" and "Notification of completion" ones.

The User Front End is composed of different components which relations are explained in the below diagrams:
**Backend Engine**

The Back End module is responsible for the execution of the request from the user and the notification of the results. It allows running the chosen pipeline with the user specified parameters.

Architecturally the Back End Engine might be described like a specialization of the *Systematic Product Generation Application* without the Queue Manager and with only a single queue of requests coming from the user. For the On Demand Processing, the Engine skips the pre-processing and post-processing phases, and executes directly the processing phase of the *Product Generation Framework*. This implies that the selected observation needs to have been run already by the SPG, so the OPG can start with the corresponding Level0 products.

The communication between the two modules is based on messages in XML which specify the available options and the values decided by the user.

The main XML message is a formal description of the Signature of the Pipeline script as described at *Pipeline Framework*. The XML description is used to visualize and validate the choices of the user. Once the inputs are validated and confirmed by the user, their values are stored back into the same XML format and delivered back to the Back End Engine, which uses the message to store a logical request of execution in the request queue and to preserve the user’s options.
At the end of the execution, the Back End module stores the results and notifies the Front End module about the completion of the request, so it in turn may inform the user.
Bulk Product generation

Bulk Product Generation is done during routine and post operation phases of the mission to produce a new version of the standard products and store them in the archive. It's estimated that this will occur once every two or three years as better data process software and calibration data become available. The same infrastructure used for Standard Product Generation is used for BPG.
Quick Look Analysis

Quick Look Analysis is an application that monitors an instrument's status and provides the opportunity for a quick assessment of the success or failure of an observation. It monitors the basic parameters located mainly in the science telemetry packets.

QLA is a GUI application that makes use of the DP and HCSS framework. It connects to the router which routes telemetry packets from the instrument/spacecraft to any application that connects to it. QLA then processes the stream of packets it receives from the router. It opens any science data packets and does simple processing such as background subtraction or averaging. The result is uncalibrated science data which is displayed in an instrument specific way.